Sumário

1
Introdução

Qualquer pessoa que utilize um computador atualmente sabe que existe algo chamado **sistema operacional**, que de alguma forma controla o equipamento. Isso é válido para qualquer tipo de computador. Aplica-se tanto ao microcomputador que é utilizado em casa quanto ao computador de grande porte utilizado pela universidade. Entretanto, é necessário definir melhor o que é um sistema operacional. Neste capítulo, nós iremos apresentar o conceito e a funcionalidade básica de um sistema operacional. Detalhes específicos serão apresentados no decorrer deste livro.

1.1 Conceito básico

Em torno de um computador, existem usuários com problemas para serem resolvidos. Por exemplo, um usuário precisa editar texto, enquanto outro precisa fazer a contabilidade da empresa. O problema de cada usuário será resolvido por um programa específico. No exemplo, um editor de textos e um sistema de contabilidade. O dispositivo físico capaz de executar esses programas é o hardware do computador.

Os programas possuem muito em comum. Por exemplo, tanto o editor de texto como a contabilidade precisam acessar o disco. A forma de acesso aos periféricos é a mesma para todos os programas. Para um melhor aproveitamento do hardware, vários usuários compartilham simultaneamente o computador. Entretanto, os programas podem apresentar necessidades conflitantes, pois disputam os recursos do equipamento. Por exemplo, o editor de texto e a contabilidade podem querer utilizar, ao mesmo tempo, a única impressora disponível.

O **sistema operacional** é uma camada de software colocada entre o hardware e os programas que executam tarefas para os usuários. Essa visão de um sistema computacional é ilustrada na Figura 1.1. O sistema operacional é responsável pelo acesso aos periféricos. Sempre que um programa necessita de algum tipo de operação de entrada e saída, ele a solicita ao sistema operacional. Dessa forma, o programador não precisa conhecer os detalhes do hardware. Informações do tipo "como enviar um caractere para a impressora" ficam escondidas dentro do sistema operacional. Ao mesmo tempo, como todos os acessos aos periféricos são feitos através do sistema operacional, ele pode controlar qual programa está acessando qual recurso. É possível, então, obter uma distribuição justa e eficiente dos recursos. Por exemplo, a divisão do espaço em disco entre os usuários é feita pelo sistema operacional. Ela pode ser feita, considerando-se dois aspectos: a eficiência no acesso ao disco e a ocupação equilibrada do disco pelos usuários.

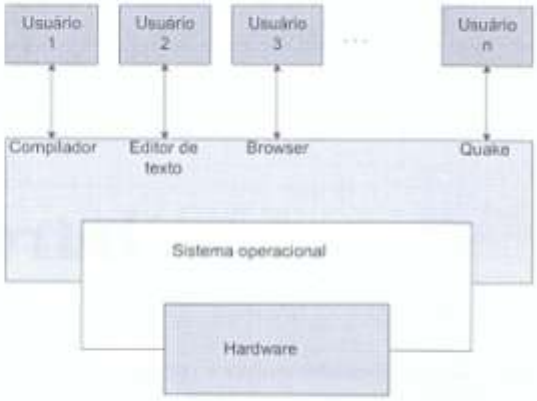

Figura 1.1 - Sistema computacional.

1.2 Objetivos do sistema operacional

O sistema operacional procura tornar a utilização do computador, ao mesmo tempo, mais eficiente e mais conveniente. A utilização mais eficiente busca um maior retorno no investimento feito no hardware. Maior eficiência significa mais trabalho obtido do mesmo hardware. Uma utilização mais conveniente vai diminuir o tempo necessário para a construção dos programas. Isso também implica a redução no custo do software, pois são necessárias menos horas de programador.

Uma utilização mais eficiente do computador é obtida através da distribuição de seus recursos entre os programas. Neste contexto, são considerados recursos quaisquer componentes do hardware disputados pelos programas. Por exemplo, espaço na memória principal, tempo de processador, impressora, espaço em disco, acesso a disco, etc.

Uma utilização mais conveniente do computador é obtida, escondendo-se do programador detalhes do hardware, em especial dos periféricos. Por exemplo, para colocar um caractere na tela do terminal, em geral é necessário toda uma seqüência de acessos à interface do terminal. Diversos registradores de controle e de *status* devem ser lidos ou escritos. Além disso, pode haver mais de um tipo de interface, com diferentes seqüências de acesso. Ao usar o sistema operacional, o programador apenas informa qual caractere deve ser colocado na tela. Todo o trabalho de acesso ao periférico é feito pelo sistema operacional.

Ao esconder os detalhes dos periféricos, muitas vezes são criados recursos de mais alto nível. Por exemplo, os programas utilizam o espaço em disco através do conceito de arquivo. Arquivos não existem no hardware. Eles formam um recurso criado a partir do que o hardware oferece. Para o programador, é muito mais confortável trabalhar com arquivos do que receber uma área de espaço em disco que ele próprio teria que organizar.

1.2.1 Tipos de serviços

Para atingir os objetivos propostos, o sistema operacional oferece diversos tipos de serviços. A definição precisa dos serviços depende do sistema operacional em consideração. Entretanto, a maioria dos sistemas operacionais oferece um conjunto básico de serviços, sempre necessários. Essa seção descreve esse conjunto básico.

Todo sistema operacional oferece meios para que um programa seja carregado na memória principal e executado. Em geral, um arquivo contém o programa a ser executado. O sistema operacional recebe o nome do arquivo, aloca memória para o programa, copia o conteúdo do arquivo para a memória principal e inicia sua execução. Também é possível abortar a execução de um programa. Isso é necessário quando, por exemplo, um programa em teste entra em um laço infinito. Nesse caso, o sistema precisa da indicação "qual programa deve ser abortado". Pode-se fazer com que cada programa esteja associado a um terminal. O programa a ser abortado é aquele associado ao mesmo terminal onde foi dada a ordem. A ordem, no caso, é o acionamento de alguma combinação especial de teclas. Pode-se também associar um número a cada programa. Nesse caso, o sistema operacional recebe o número do programa a ser abortado.

Talvez o serviço mais importante oferecido pelo sistema operacional seja o que permite a utilização de arquivos, serviço este implementado através do sistema de arquivos. Através dele, é possível criar, escrever, ler e destruir arquivos. Através da leitura e escrita, é possível copiar, imprimir, consultar e atualizar arquivos. Em geral, também existem operações do tipo renomear, obter o tamanho, obter a data de criação e outras informações a respeito dos arquivos.

Todo acesso aos periféricos é feito através do sistema operacional. Na maioria das vezes, os discos magnéticos são acessados de forma indireta, através dos arquivos. Entretanto, muitos dispositivos podem ser acessados de forma direta. Entre eles, estão os terminais, impressoras, fitas magnéticas e linhas de comunicação. Para tanto, devem existir serviços do tipo alocação de periférico, leitura, escrita e liberação. Em geral, também pode-se obter informações a respeito de cada periférico e alterar algumas de suas características. Por exemplo, alterar a velocidade de uma linha de comunicação.

À medida que diversos usuários compartilham o computador, passa a ser interessante saber quanto de quais recursos cada usuário necessita. Pode-se utilizar essa informação para calcular o valor a ser cobrado pelo uso do computador. Mesmo que não exista um sistema de cobrança, a monitoração do uso dos recursos pode levar à identificação de gargalos dentro do sistema. Um exemplo de gargalo seria pouca memória principal, ou ainda um disco muito lento. Um serviço oferecido por muitos sistemas operacionais é a contabilização do uso dos recursos pelos programas e usuários. Esse serviço fornece estatísticas do tipo "quando tempo de processador foi gasto na execução de um programa", "qual o espaço em disco ocupado pelos arquivos de um determinado usuário" ou "quantas páginas foram impressas este mês por determinado usuário".

Diversas informações sobre o estado do sistema são mantidas pelo sistema operacional. Em geral, essas informações são necessárias para o próprio funcionamento do sistema. Entretanto, elas também podem ser fornecidas aos programas e usuários. Dessa forma, a manutenção dessas informações pode ser considerada como mais um serviço que é oferecido. 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, entre outros dados.

Na busca de um melhor aproveitamento do hardware, diversos usuários podem compartilhar um computador. Entretanto, isso somente é viável se houver algum tipo de proteção entre os usuários. Não é aceitável, por exemplo, que um usuário envie dados para a impressora no meio da listagem

de outro usuário. Outro exemplo de interferência entre usuários seria a destruição de arquivos ou o cancelamento da execução do programa de outra pessoa. Não é possível contar apenas com a correção dos programas e a boa intenção dos usuários para tratar desse problema. Se não houver segurança com respeito à execução dos programas e à manutenção dos dados, os usuários simplesmente não utilizarão o computador. Cabe ao sistema operacional garantir que cada usuário possa trabalhar sem sofrer interferência danosa dos demais. Pode-se, então, considerar como um serviço do sistema operacional a criação de mecanismos de proteção entre os usuários.

1.3 Sistema operacional na visão do usuário

A arquitetura de um sistema operacional corresponde à imagem que o usuário tem do sistema, a forma como ele percebe o sistema. Essa imagem é definida pela interface através da qual o usuário acessa os serviços do sistema operacional. Essa interface, assim como a imagem, é formada pelas chamadas de sistema e pelos programas de sistema.

1.3.1 Chamadas de sistema

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. São as sub-rotinas da biblioteca que chamam o sistema. Por exemplo, qualquer função da biblioteca que acesse o terminal (como `printf()` na linguagem C) exige uma chamada de sistema. Como foi exposto antes, o acesso aos periféricos é feito, normalmente, pelo sistema operacional.

A lista de serviços do sistema operacional é agora transformada em uma lista de chamadas de sistema. A descrição dessas chamadas forma um dos mais importantes manuais de um sistema operacional. Por exemplo, considere um programa que lista o conteúdo de um arquivo texto na tela do terminal. Ele faz uma chamada de sistema para verificar se o arquivo a ser listado existe. Um dos parâmetros dessa chamada será provavelmente o nome do arquivo a ser listado. O restante do programa é um laço no qual são feitas sucessivas leituras do arquivo e escritas no terminal. Essas duas operações também correspondem a chamadas 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 visto com detalhes nos próximos capítulos.

Em função da complexidade interna de um *kernel* completo, muitos sistemas operacionais são implementados em camadas. Primeiro, um pequeno componente de software chamado **micronúcleo** ou ***microkernel*** implementa os serviços mais básicos associados com sistemas operacionais. Em cima do *microkernel*, usando os seus serviços, o *kernel* propriamente dito implementa os demais serviços.

A Figura 1.2 ilustra um sistema no qual, acima do hardware, existe um ***microkernel*** que oferece serviços básicos tais como gerência do processador, alocação e liberação de memória física e instalação de novos tratadores de dispositivos. O *kernel* do sistema oferece serviços tais como sistema de arquivos, memória virtual e protocolos de comunicação.

Alguns sistemas permitem que as aplicações acessem tanto as chamadas de sistema suportadas pelo *kernel* quanto os serviços oferecidos pelo *microkernel*. Entretanto, na maioria das vezes, apenas o código do *kernel* pode acessar os serviços do *microkernel*, enquanto aplicações ficam restritas às chamadas de sistema do *kernel*.

Figura 1.2 – Organização do sistema em *kernel* e *microkernel*.

1.3.2 Programas de sistema

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. Como implementam tarefas essenciais para a utilização do computador, são, em geral, distribuídos pelo próprio fornecedor do sistema operacional.

Exemplos são os utilitários para manipulação de arquivos: programas para listar arquivo, imprimir arquivo, copiar arquivo, trocar o nome de arquivo, listar o conteúdo de diretório, entre outros. Esses utilitários são, em geral, programas normais. Eles utilizam chamadas de sistema para efetuar a operação solicitada pelo usuário. Também é usual o emprego de programas de sistema para a obtenção de informações a respeito do sistema, tais como data, hora ou quais usuários estão utilizando o computador no momento.

Na década de 1960, compiladores eram também considerados programas de sistema, fornecidos junto com o sistema operacional. Entretanto, com a expansão da indústria de microcomputadores, compiladores passaram a ser considerados programas normais. É usual o fornecedor do sistema operacional também oferecer uma linha de compiladores, ou fornecer alguns junto com o sistema. Mas, como diversos fornecedores existem, pode-se comprar o sistema operacional de uma empresa e o compilador de outro. O mesmo é válido para montadores e editores de texto.

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. Isso é feito através de uma chamada de sistema. A informação pode ser formatada para facilitar a sua disposição na tela do terminal. Finalmente, novas chamadas de sistema serão feitas para listar essas informações na tela.

Observe que a operação "lista diretório" pode ser feita através do interpretador de comandos ou de um utilitário. Em qualquer situação, as mesmas chamadas de sistema estarão envolvidas. É uma questão de projeto decidir quais comandos serão aceitos pelo interpretador. Os demais serão implementados através de utilitários. Em geral, comandos aceitos pelo interpretador são executados mais rapidamente, pois não exigem a carga e a execução de um outro programa. No caso, o outro programa é o utilitário. Entretanto, um interpretador de comandos muito grande pode tornar-se complexo. Além disso, ele vai ocupar memória com o código necessário para executar comandos que raramente são chamados.

Em determinados momentos, o interpretador de comandos deve disparar a execução de utilitários, ou mesmo de programas do usuário. Nesse caso, ele deve utilizar uma chamada de sistema que permita a um programa disparar a execução de outro programa.

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: pode-se alterar o interpretador de comandos sem mexer no sistema operacional. Mais ainda, pode-se construir vários interpretadores de comandos. Nesse caso, cada usuário utilizaria um interpretador de comandos apropriado a sua experiência ou trabalho a ser desenvolvido.

Tudo 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 única diferença está na comodidade para o usuário, que passa a usar ícones, menus e mouse no lugar de digitar comandos textuais.

Na maioria das vezes, o interpretador de comandos, ou a GUI correspondente, "é o sistema operacional" para o usuário. Ele será julgado pela sua facilidade de uso em um primeiro momento. À medida que os usuários ganham experiência com o sistema, passam a julgar as interfaces pela sua flexibilidade e capacidade de realizar tarefas mais complexas.

Na maior parte do tempo, o usuário trabalha com programas distantes do sistema operacional. Programadores utilizam principalmente editores de texto e compiladores. Usuários finais utilizam aplicativos e ferramentas de apoio. A opinião do usuário a respeito do sistema como um todo vai depender, principalmente, desses programas. O sistema operacional propriamente dito fica escondido, longe da percepção do usuário comum.

1.4 Sistema operacional na visão de projeto

Na visão de projeto, o mais importante é como o sistema está organizado internamente. A organização de um sistema operacional corresponde à forma como ele implementa os vários serviços.

O sistema operacional não resolve os problemas do usuário final. Ele não serve para editar texto, nem faz a contabilidade da empresa. Entretanto, através dele, podemos obter uma maior eficiência e conveniência no uso do computador. A eficiência é obtida através do compartilhamento dos recursos. A conveniência é obtida através de uma interface mais confortável para a utilização dos recursos computacionais.

Normalmente, o processador está executando programas de usuário. Para isso que o computador foi comprado. Somente quando ocorre algum evento especial, o sistema operacional é ativado. Dois tipos de eventos ativam o sistema operacional: uma chamada de sistema ou uma interrupção de periférico.

Uma chamada de sistema corresponde a uma **solicitação de serviço** por parte do programa em execução. Primeiramente, deve ser verificada a legalidade da solicitação. Por exemplo, um pedido para que arquivos de outros usuários sejam destruídos deverá ser recusado. No caso de uma solicitação legal, ela é realizada, e a resposta é devolvida ao programa. É possível que a chamada de sistema envolva o acesso a um periférico. Nesse caso, o programa deverá esperar até que o periférico conclua a operação solicitada.

Em função das chamadas de sistema, o sistema operacional envia comandos para os controladores dos periféricos. O controlador deve informar ao sistema operacional quando a operação estiver concluída. Isso é feito através de uma interrupção. Quando a interrupção acontece, o processador pára o que está fazendo e passa a executar uma rotina específica do sistema operacional. Como a interrupção do periférico avisa o término de alguma operação de entrada e saída, possivelmente uma chamada de sistema foi concluída. Nesse caso, um programa à espera de resposta poderá ser liberado.

Por exemplo, considere um programa que faz uma chamada de sistema para ler um registro de arquivo em disco. O sistema operacional envia o comando para o controlador do disco. O programa fica parado à espera da resposta. Quando a leitura é concluída, o controlador do disco informa ao sistema operacional, que então libera o programa do usuário para prosseguir sua execução.

A descrição de sistema operacional apresentada nessa seção deixa clara a importância das interrupções para a sua construção. Esse é possivelmente o mecanismo de hardware mais importante para a construção de um sistema operacional moderno, juntamente com o conceito de unidade de gerência de memória (MMU- *Memory Management Unit*). A função da MMU será descrita no capítulo que trata da gerência de memória. Os capítulos sobre multiprogramação e sobre gerência de entrada e saída descreverão o funcionamento do mecanismo de interrupção.

1.5 Histórico de sistemas operacionais

Nessa seção, será feito um rápido histórico da evolução dos sistemas operacionais através do tempo. As datas fornecidas para cada período são aproximadas. O objetivo é apenas posicionar o leitor no tempo. Não existe a pretensão de definir com exatidão o momento em que determinada técnica surgiu ou passou a ser utilizada. Mesmo porque, na rápida evolução da computação, as etapas acabam ficando sobrepostas umas às outras. As novas técnicas não são assimiladas simultaneamente por todas as instalações. Muito pelo contrário, sempre existe alguma inércia contra qualquer alteração na forma de operação do computador.

Nos primórdios da computação, na década de 40, não existia sistema operacional. Nesse ambiente, o programador é também o operador do computador. Existe uma planilha para alocação de horário na máquina. Durante o seu horário, o programador controla todo o equipamento. Um programa, quando executado, tem controle total da máquina. O programa acessa diretamente aos periféricos. No máximo, existe uma biblioteca com rotinas de entrada e saída já programadas.

A primeira modificação introduzida nesse esquema foi a utilização de operadores profissionais. O programador não mais opera o computador durante a execução de seu programa. Ele entrega ao operador o seu *job*. O ***job*** é formado pelo programa a ser compilado e executado, acompanhado dos dados para a execução. Geralmente, os programas e dados são preparados na forma de cartões perfurados. Após a execução, o programador recebe uma listagem com a saída gerada pelo programa. No caso de erro durante a execução, o operador pode também tirar uma cópia na listagem do conteúdo de toda a memória principal. Isso é possível pois, nessa época, a

memória principal não passa de poucos kilobytes. A depuração do programa deve ser feita, pelo programador, a partir dessas listagens.

O emprego de operadores profissionais diminui o tempo que o computador fica parado. Não existem as perdas inerentes a uma planilha de alocação de horário. Enquanto o programador pensa a respeito do que saiu errado, outro programa está sendo executado. Entretanto, o tempo de preparação para a execução de um *job* continua grande. É necessário primeiro retirar as fitas magnéticas, cartões e listagens do *job* que terminou. Depois, são preparadas as fitas magnéticas e os cartões do próximo *job*. Para diminuir esse tempo entre *jobs*, eles passam a ser agrupados em lotes. Em um mesmo **lote**, ou ***batch***, são colocados *jobs* com necessidades semelhantes. Por exemplo, todos os que irão usar o mesmo montador e a mesma biblioteca. Dessa forma, pouca preparação é necessária entre *jobs* de um mesmo lote. Essa é a origem do termo **sistema em *batch***.

Mesmo com o agrupamento dos *jobs* semelhantes em lotes, a passagem entre *jobs* continua sendo feita de forma manual. O operador precisa ficar atento à console do computador. Quando o programa em execução terminar, ele deve manualmente comandar a carga e a execução do próximo programa. Um mesmo *job* pode exigir a execução de vários programas. Por exemplo, primeiro o montador é executado e traduz o programa em *assembly* para linguagem de máquina. Depois, o programa montado é ligado com as rotinas de entrada e saída de uma biblioteca. Finalmente, o programa já pronto é carregado e executado. Cada uma dessas etapas é chamada de ***step***. Logo, um *job* pode ser formado por vários *steps*.

Na década de 1950, surgiram os primeiros monitores residentes. Sua função é automatizar a transição do computador entre programas. O **monitor residente** é um programa que fica o tempo todo na memória. Quando um programa em execução termina, ele avisa o monitor. O monitor residente então automaticamente carrega o próximo programa e inicia a execução dele. O tempo em que o computador fica parado diminui. A transição entre programas feita pelo monitor residente é mais rápida, pois dispensa a operação manual.

Para que o monitor residente consiga substituir o operador na transição entre *jobs*, ele precisa saber o que fazer. Ele precisa saber qual programa deve ser carregado e executado a seguir. São os **cartões de controle** que fornecem essa informação. Eles são diferenciados de cartões normais, com programa ou dados, por algum símbolo. Como, por exemplo, duas barras no início do cartão. Além de informar ao monitor residente o que fazer a seguir, eles podem ser utilizados para outras informações. Por exemplo, o cartão inicial do *job* pode conter a identificação do usuário para o qual será contabilizado o custo de processamento. Pode também trazer um tempo máximo de execução, após o que o programa deve ser considerado com erro e abortado. Na verdade, os cartões de controle são a origem das linguagens de comandos utilizadas em sistemas atuais.

O monitor residente também é o local indicado para as rotinas de acesso aos periféricos. São rotinas utilizadas por todos os programas. Com elas no monitor residente, as aplicações não precisam acessar diretamente os periféricos. Apenas chamar a rotina apropriada dentro do monitor. Esse é o início da idéia de chamada de sistema. Além de simplificar os aplicativos, essa técnica permite uma maior flexibilidade. Por exemplo, se um periférico é trocado, novas rotinas de acesso são escritas e colocadas no lugar das anteriores. Entretanto, os aplicativos não sofrem alteração. Eles continuam apenas chamando o monitor residente para fazer entrada e saída.

Na década de 1960, a partir do monitor residente, surgiu o conceito de **multiprogramação**. No monitor residente, apenas um programa é executado de cada vez. Quando ele precisa fazer alguma entrada e saída, o processador fica parado. Em geral, periféricos são dispositivos eletromecânicos e trabalham na faixa de milisegundo. Ao mesmo tempo, o processador é um dispositivo eletrônico, que trabalha na faixa de microsegundo. Por exemplo, enquanto é feito um único acesso à leitora

de cartões, poderiam ser executadas 10000 instruções de máquina ou mais. Em conseqüência dessa diferença de velocidade, a utilização do processador é muito baixa. Durante a maior parte do tempo, o programa está parado, esperando o término de uma operação de entrada ou saída.

A solução imaginada foi manter diversos programas na memória principal ao mesmo tempo. Quando um dos programas está esperando a conclusão da entrada ou saída, outro programa inicia sua execução. Quando o primeiro termina o acesso ao periférico, podemos retomar sua execução, ou esperar que esse segundo programa solicite também alguma entrada ou saída, para então suspender sua execução. Agora o tempo do processador é dividido entre diversos programas. O objetivo é ocupar melhor o hardware. Com multiprogramação, o processador fica menos tempo parado. Os periféricos também são melhor utilizados, pois, com um número maior de programas em execução, as solicitações serão mais freqüentes.

Duas inovações de hardware possibilitaram o desenvolvimento da multiprogramação. Em primeiro lugar, o uso de **interrupções**. Quando um comando é enviado para um periférico, imediatamente o sistema operacional inicia a execução de um outro programa. É necessário que o periférico avise ao sistema operacional quando o acesso tiver sido concluído. Isso é feito através de uma interrupção. Em um sistema no qual apenas um programa ocupe o computador de cada vez, interrupções não são essenciais. Pode-se fazer com que o processador fique em um laço, consultando a interface do periférico, até que a operação esteja concluída. Essa técnica é chamada de ***polling*** ou ***busy-loop***. Entretanto, não é possível implementar multiprogramação sem o auxílio de interrupções.

O desenvolvimento dos **discos magnéticos** também foi importante para o surgimento da multiprogramação. Os *jobs* eram submetidos em cartões perfurados. Inicialmente, eles eram lidos pelo computador diretamente dos cartões. Mais tarde, eles passaram a ser antes copiados para fita magnética. O computador passou então a ler os *jobs* de fita magnética, e não de cartões. Como a fita é mais rápida, esse mecanismo diminuiu o tempo de entrada e saída e, por conseqüência, o tempo de processador parado. Entretanto, leitoras de cartões e unidades de fita magnética são dispositivos essencialmente seqüenciais. Somente é possível ler o segundo *job* depois de ler completamente o primeiro *job*. Na multiprogramação, o segundo *job* é necessário antes do término do primeiro. Por exemplo, quando o primeiro *job* solicita uma operação de entrada para ler um dos seus cartões de dados, é necessário ter na memória principal o programa a ser executado para o segundo *job*. Isso significa ler cartões do segundo *job* sem ter lido ainda todos os cartões do primeiro *job*. Em um dispositivo seqüencial, isto não é possível.

O disco magnético permite a implementação da multiprogramação. Vários *jobs* são lidos de cartão perfurado ou fita magnética para o disco. Como o disco permite um acesso direto a qualquer posição, é possível ler parte do primeiro *job* e do segundo *job*. À medida que, em função de sua execução, um *job* solicita a leitura de mais dados, eles são buscados no disco. A execução de vários *jobs* pode agora ser sobreposta sem problemas.

O conceito de multiprogramação é essencial no estudo de sistemas operacionais. Ele também é a origem da maioria dos temas básicos da área. Por exemplo, agora que existem vários *jobs* na memória principal, existe a necessidade de controlar a ocupação da memória. Ao mesmo tempo, é preciso escolher qual *job* executar a seguir. Também é necessário organizar o conteúdo do disco, para que a informação desejada seja rapidamente localizada. Ao longo do texto, o conceito de multiprogramação estará sempre presente.

Com a utilização de disco magnético, não havia mais a necessidade de reunir *jobs* semelhantes em lotes. O termo ***batch*** passou então a designar um sistema no qual não existe interação entre o usuário e a execução do programa. Mais modernamente, o termo *batch* foi substituído por "**execução em *background***". A depuração de programas em um ambiente *batch* é difícil. O

sistema operacional para facilitar essas três tarefas. Uma vantagem do suporte a clustering no Windows 2000 é que ele foi concebido de forma a não necessitar de nenhum tipo especial de plataforma ou de conectividade. Um conjunto de máquinas interconectados normalmente em rede podem compor um *cluster*.

10.10 Uma palavrinha sobre o Windows XP

O Windows XP é a geração seguinte da família Windows. A denominação XP vem da palavra *eXPerience*. O Windows XP foi idealizado pela Microsoft com o objetivo de unificar, em torno de um único produto, seu mercado corporativo com seu mercado de usuários domésticos. Na realidade, essa unificação é feita através de duas versões do Windows XP: o *Windows XP Personal Edition*, destinado ao mercado doméstico, que substitui o Windows 95, 98, Millennium, NT (versão workstation); e o *Windows XP Professional Edition*, voltado ao mercado corporativo que substitui o NT nas suas versões *server*.

As principais novidades introduzidas pelo Windows XP estão relacionadas com mecanismos de proteção ao sistema de arquivos e conectividade à Internet. Sob o ponto de vista de proteção do sistema de arquivos, o Windows XP impede que arquivos antigos substituam versões mais recentes. No caso de drivers de dispositivos, é possível restaurar a versão anterior na eventualidade da instalação de um driver mais recente apresentar problemas. O suporte à proteção, na presença de múltiplos usuários, foi estendido em relação aos mecanismos oferecidos pelo Windows 98 e pelo Windows Millenium embutidos em seu próprio núcleo. Em relação à conectividade em rede, quando conectado à Internet, o Windows XP oferece ao usuário algumas funcionalidades típicas de *firewall* embutidas em seu próprio núcleo. Além disso, uma série de ferramentas buscando simplificar o uso do Windows por usuário leigos foram introduzidas, como por exemplo, gravação de cdrom diretamente a partir do *Windows Explorer*, emprego de temas (*skins*) para tela de fundo, atualizações automáticas, mecanismos para publicação de arquivos de imagens e de texto na Internet, etc.

Entre as novidades do Windows XP, está ainda um mecanismo de proteção contra pirataria denominado de WPA (*Windows Product Activation*). Seu funcionamento é baseado na criação de um código único, válido apenas para o computador no qual o Windows XP é instalado. Esse código é criado no momento da instalação e é obtido através de identificadores próprios únicos a cada computador, tais como número de série da BIOS, do disco rígido, o endereço físico da placa de rede (endereço MAC), etc. Esse código é então informado à Microsoft, que imediatamente gera e reenvia um código de liberação para o uso do Windows XP. Ambos os códigos, o gerado na instalação e o código de instalação, são cadastrados na Microsoft. Sempre que houver uma modificação de hardware da máquina, ou a tentativa de burlar o sistema de ativação do Windows XP, o usuário deverá repetir o procedimento de ativação do Windows XP, ou seja, recontactar a Microsoft e gerar uma nova chave de ativação. Esse procedimento criou, na comunidade de usuários, muita controvérsia devido a rumores de que a Microsoft aproveitava-se dele para obter informações adicionais sobre a configuração da máquina, como por exemplo, os softwares instalados. Essa polêmica levou à análise das transações realizadas pelo mecanismo WPA por consultores independentes que concluíram que apenas o código de ativação é enviado à Microsoft.

O Windows XP foi desenvolvido com a preocupação de manter a compatibilidade com várias aplicações já existentes para a família Windows, principalmente jogos e multimídia. A Microsoft divulga que todos os aplicativos existentes para Windows 98, Millenium, e Windows NT continuarão a funcionar normalmente no Windows XP. A mesma preocupação de compatibilidade

existe a nível de hardware; assim sendo, o Windows XP dispõe de uma grande gama de drivers para os mais diversos periféricos. Além disso, é possível instalar em uma máquina Windows XP drivers existentes para outras versões da família Windows.

As diferenças entre as versões Windows XP *Personal Edition* e Windows XP *Professional Edition* estão relacionadas com desempenho e atividades de gerenciamento. A versão *Professional* explora o multiprocessamento real oferecido pelas máquinas multiprocessadoras, ao passo que a versão *Personal* é otimizada para máquinas monoprocessadoras.

Sob o ponto de vista do sistema operacional, o Windows XP é um sistema operacional de 32 bits e herda em muito a arquitetura NT 5.0. As modificações, segundo a Microsoft, estão em otimizações de algoritmos básicos e nas estruturas de dados internas ao núcleo. Essas melhorias fazem com que o Windows XP apresente um desempenho melhor que seus antecessores. O Windows XP, assim como o Windows 2000 (NT 5.0), não fornece a capacidade de realizar *boot* em modo DOS. A compatibilidade com aplicativos que rodam sob DOS (como o Clipper) é feita exatamente da mesma forma que no Windows 2000, ou seja, através de um emulador DOS.

10.11 Exercícios

1. Compare o sistema de paginação do Windows com o do Linux, considerando os aspectos: políticas de substituição de páginas, tradução de endereço lógico a endereço físico, e estratégia de alocação de páginas em memória. (Seção 10.5)
2. O Linux utiliza um modelo de estados de processo diferente do Windows. Faça uma correspondência entre os estados utilizados em cada um desses sistemas operacionais. (Seção 10.4)
3. Pesquise sobre a capacidade do Linux de atribuir prioridades para *threads* no mesmo estilo da classe de tempo real oferecida pelo Windows. Caso exista algum mecanismo, faça uma análise comparativa. (Seção 10.4)
4. Cite prós e contras do sistema de *swapping* utilizado pelo Windows quando confrontado com o do Linux. (Seção 10.5)
5. Analise como é organizado o sistema de arquivos NTFS. Compare-o com o do Linux. (Seção 10.6

programador submete o *job* e, algumas horas depois, recebe uma listagem com o resultado da execução. Os erros do programa devem ser detectados e corrigidos nesse contexto.

Durante a década de 60, iniciaram as primeiras experiências com sistemas ***timesharing***. Na década de 1970, ocorreu sua disseminação. Em um ambiente com multiprogramação, diversos programas dividem o tempo do processador. Em um sistema *timesharing*, além da multiprogramação, cada usuário possui um terminal. Através desse terminal, o usuário pode interagir com o programa em execução. Por exemplo, um programador pode acompanhar passo a passo a execução de um programa em depuração. Ao detectar um erro, ele corrige e já inicia uma nova execução. A depuração é muito mais rápida do que em um ambiente *batch*.

Cada usuário, em seu terminal, tem a sensação de possuir o computador apenas para seus programas. Esse compartilhamento é possível, pois usuários em terminais consomem pouco tempo de processador. Enquanto o usuário pensa, conversa ou toma café, ele não está ocupando o processador. Isso acontece somente quando seus programas executam. Mesmo assim, programas em ambiente *timesharing* usualmente são bastante interativos. Por exemplo, considere um editor de texto, no qual um programa está sendo digitado. No intervalo entre o acionar das teclas, o programa não gasta tempo de processador. Ele está parado, aguardando a próxima tecla. Ao receber o caractere digitado, o programa editor de texto armazena-o nas suas estruturas de dados. Ele então volta a ficar bloqueado, à espera do próximo caractere. Em termos de instruções de máquina, o tempo entre a digitação de dois caracteres é muito grande, permitindo a execução de milhões de instruções de máquina (os digitadores mais rápidos possuem velocidade de 5 caracteres por segundo).

A computação está em permanente evolução. Por exemplo, na década de 1980, temos uma enorme disponibilidade de microcomputadores. No início, os sistemas operacionais para microcomputadores eram bastante simples, lembrando até o velho monitor residente. À medida que o hardware dessas máquinas foi melhorado, o mesmo aconteceu com o sistema operacional. Atualmente, temos em microcomputadores mecanismos encontrados, algum tempo atrás, apenas em máquinas de grande porte.

Historicamente, sistemas operacionais preocuparam-se principalmente com a eficiência no uso do computador. Atualmente, existe uma grande preocupação com a conveniência no uso. Isso em parte é conseqüência da redução no custo dos equipamentos, se comparado com o custo de pessoal. Essa preocupação com conveniência deu origem, por exemplo, a interfaces mais amigáveis. A interface tradicional do interpretador de comandos é bastante dura, baseada em comandos mnemônicos e em seus parâmetros, os quais devem ser memorizados pelo usuário. Atualmente, existe grande ênfase em interfaces baseadas em menus de comandos, mouse para indicação de comandos, janelas para acompanhar a execução de vários comandos simultaneamente e ícones. Tudo isso para simplificar a utilização do computador.

A maior parte das aplicações comerciais, mesmo em microcomputadores, são hoje construídas em torno de **bancos de dados**. Ainda que a maioria dos sistemas gerenciadores de bancos de dados sejam implementados fora do sistema operacional, este deve oferecer algum suporte. Por exemplo, vários programas podem querer acessar o banco de dados ao mesmo tempo. O sistema operacional pode, nesse caso, oferecer mecanismos para o compartilhamento controlado de arquivos.

Uma das áreas de pesquisa mais importantes atualmente são os **sistemas operacionais distribuídos**. Em um sistema desse tipo, vários computadores estão interconectados através de uma rede de comunicação de algum tipo. É possível, a partir de um dos computadores, acessar recursos em outros. Por exemplo, arquivos e periféricos de outro computador, ou até mesmo enviar um programa para ser executado em uma máquina de menor carga no momento. A

gerência dessas solicitações aumenta a complexidade do sistema operacional. Ao mesmo tempo, cria sérios problemas de segurança. Em um sistema distribuído, fica mais difícil o controle de acessos não autorizados. Com o surgimento da **Internet**, a importância dos sistemas distribuídos aumentou ainda mais.

Outra área importante são os **sistemas operacionais de tempo real**, usados no suporte às aplicações submetidas a requisitos de natureza temporal. Nesses sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Aplicações com requisitos de tempo real são cada vez mais comuns. Essas aplicações variam muito com relação ao tamanho, complexidade e "criticalidade". Entre os sistemas mais simples estão os controladores embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade desse espectro estão os sistemas militares de defesa e o controle de tráfego aéreo. Exemplos de aplicações críticas são os sistemas responsáveis pelo monitoramento de pacientes em hospitais e os sistemas embarcados em veículos, de automóveis até aviões e sondas espaciais. Entre aplicações não críticas estão os videogames e as aplicações multimídia em geral.

À medida que são desenvolvidos computadores com diversos processadores, passa a ser necessário rever aspectos básicos dos sistemas operacionais. Por exemplo, agora existe um **paralelismo** real a ser aproveitado. Diversos programas podem ser executados realmente em paralelo. É necessário agora decidir em que processador será executado qual programa. Além disso, o que essa escolha vai implicar em termos de gerência de memória. Essa questão é importante pois nem todos os processadores possuem acesso a toda a memória. Junto com sistemas distribuídos, sistemas operacionais para arquiteturas paralelas são uma das áreas de maior pesquisa atualmente em sistemas operacionais.

1.6 Exercícios

1) Consulte os manuais do sistema operacional usado na sua empresa ou universidade e liste os diferentes tipos de chamadas de sistema existentes.

2) Classifique as chamadas de sistema levantadas no exercício anterior conforme os tipos de serviço apresentados na Seção 1.2, criando novos tipos de serviço caso seja necessário.

3) Muitos sistemas operacionais atuais são projetados para operar em redes de computadores. Cite vantagens e desvantagens dessa operação em rede, quando comparada com a operação clássica (*stand-alone*).

2
Multiprogramação

Como visto no capítulo de introdução, a **multiprogramação** torna mais eficiente o aproveitamento dos recursos do computador. Isso é conseguido através da execução simultânea de vários programas. Neste contexto, são exemplos de recursos o tempo de processador, o espaço na memória, o tempo de periférico, entre outros. Este capítulo trata dos principais conceitos associados com a multiprogramação, com destaque para **processos**, **interrupções** e a **proteção entre processos**.

2.1 Mecanismo básico

Em um **sistema multiprogramado** diversos programas são mantidos na memória ao mesmo tempo. A Figura 2.1 mostra uma possível organização da memória. Nesse sistema, existem 3 programas de usuário na memória principal, prontos para serem executados. 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.

Figura 2.1 - Memória em um sistema com multiprogramação.

A maioria dos programas não precisa de toda a memória do computador. Na verdade, muitos programas ocupam uma pequena parcela da memória principal disponível. Sem multiprogramação, a memória não ocupada por um programa ficaria sem utilização. Com vários programas na memória, esse recurso também é mais bem aproveitado.

Quando termina a operação de E/S do programa 1, ele pode voltar a ocupar o processador. Entretanto, o programa 2 está sendo executado. Será necessário selecionar qual deles ficará com o processador. Algoritmos para essa situação serão apresentados no capítulo sobre gerência de processador.

2.2 O conceito de processo

Em sistemas operacionais é conveniente diferenciar um programa de sua execução. Por exemplo, o mesmo programa pode estar sendo executado por vários usuários, ao mesmo tempo. 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 bastante abstrato, mas essencial no estudo de sistemas operacionais.

Um programa é uma seqüência de instruções. É algo passivo dentro do sistema. Ele não altera o seu próprio estado. O processo é um elemento ativo. O processo altera o seu estado, à medida que executa um programa. É o processo que faz chamadas de sistema, ao executar os programas.

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.

2.3 Ciclos de um processo

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. Por exemplo, um processo para cada terminal do computador. Nesse caso, todos os processos são criados na inicialização do sistema. Eles somente são destruídos quando o próprio sistema é desligado.

Outra forma de trabalhar com os processos é associá-los a uma sessão de trabalho. Um usuário abre uma **sessão de trabalho** fornecendo ao sistema seu código de usuário e, possivelmente, uma senha. A senha é necessária para que cada usuário limite o acesso de outros usuários aos seus arquivos. O sistema operacional verifica a validade da senha e cria um processo para atender o usuário. Provavelmente, o primeiro programa a ser executado pelo processo é o interpretador de comandos. No momento em que o usuário executa o comando "fim de sessão de trabalho", o processo associado à sessão é destruído.

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** (***daemon***), 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.

Após ter sido criado, o processo passa a ocupar processador. Em determinados momentos, ele deixa de ocupar o processador para realizar uma operação de E/S. Os momentos em que um processo não está esperando por E/S, ou seja, em que ele deseja ocupar o processador, são chamados de "**ciclos de processador**". Quando o processo está esperando por uma operação de E/S, ele está em um "**ciclo de E/S**". A chamada de sistema é o evento que termina o ciclo de processador em andamento e inicia um ciclo de E/S. A conclusão da chamada de sistema faz o caminho inverso. O primeiro ciclo da vida de um processo será necessariamente um ciclo de processador. Para entrar em um ciclo de E/S é necessário executar ao menos uma instrução, no caso, a instrução que faz a chamada de sistema.

Um processo que utiliza muito processador é chamado de ***cpu-bound***. O seu tempo de execução é definido principalmente pelo tempo dos seus ciclos de processador. Por outro lado, um processo que utiliza muita E/S é chamado de ***i/o-bound*** (*input/output-bound*). Nesse caso, o tempo de execução é definido principalmente pela duração das operações de E/S. Não existe uma quantificação precisa para essas definições. Por exemplo, um processo que executa um programa de cópia de arquivo é *i/o-bound*. Ele praticamente não utiliza processador, apenas acessa disco. Já um processo que executa um programa de inversão de matriz é *cpu-bound*. Após ler alguns poucos dados, ele precisa apenas de processador. O ideal é ter no sistema uma mistura de processos *cpu-bound* com processos *i/o-bound*. Se todos os processos forem *cpu-bound*, o processador será o gargalo do sistema. Se todos forem *i/o-bound*, o processador ficará parado enquanto todos os processos tentam acessar os periféricos.

2.4 Relacionamento entre processos

Dependendo do sistema operacional considerado, pode ou não haver algum tipo de relacionamento entre os processos. Em um sistema no qual existe independência total entre processos, operações devem ser realizadas sobre cada processo individualmente. Um exemplo de operação nesse contexto é a destruição do processo.

Alguns sistemas suportam o conceito de **grupo de processos**. Por exemplo, todos os processos associados a um mesmo terminal podem formar um grupo. Se processos são criados através de chamada de sistema, pode-se considerar como pertencentes ao mesmo grupo todos os processos criados pelo mesmo processo. O conceito de grupo permite que operações possam ser aplicadas sobre todo um conjunto de processos, e não apenas sobre processos individuais. Por exemplo, os processos pertencentes a um mesmo grupo podem compartilhar os mesmos direitos perante o sistema.

Como dito antes, em muitos sistemas os processos são criados por outros processos, através de 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, por sua vez, podem criar outros processos. Essa situação é facilmente representada através de uma

árvore. Cada nodo da árvore representa um processo. Uma ligação entre dois nodos significa que o processo representado pelo nodo superior criou o processo representado pelo nodo inferior.

A Figura 2.2 ilustra a hierarquia de processos existente em um dado instante, em um sistema hipotético. O processo P1 é o processo inicial do sistema. Todos os demais foram criados a partir dele. Esse processo inicial não foi criado por chamada de sistema, mas sim durante a inicialização do sistema operacional. Os processos P2, P3 e P4 foram criados diretamente por P1, enquanto os demais foram criados por seus processos filhos.

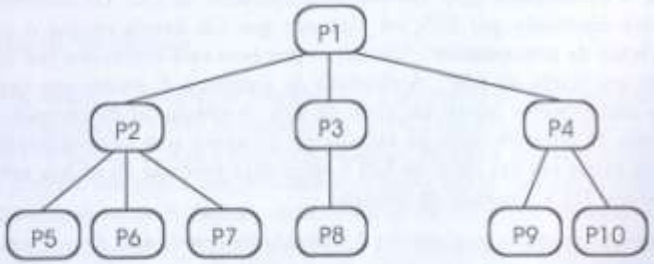

Figura 2.2 - Hierarquia de processos.

É importante salientar que o formato da árvore muda com o passar do tempo. À medida que processos são criados e destruídos, nodos são acrescentados ou removidos da árvore, respectivamente. O sistema deve definir, por exemplo, o que fazer quando um processo com filhos é destruído. Pode-se definir que, quando um processo é destruído, todos os processos que derivam dele também são destruídos. Outra solução é manter o nodo que representa o processo destruído, até que todos os seus descendentes também terminem. Uma terceira solução é vincular os processos que são filhos do processo destruído com o processo pai daquele, isso é, com o "processo avô" deles.

2.5 Estados de um processo

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. Ele precisa de processador para executar. Entretanto, o processador poderá estar ocupado com outro processo, e ele deverá esperar. Diversos processos podem estar nesse mesmo estado. Por exemplo, imagine que o processo 1 está acessando um periférico. O processo 2 está executando. Quando termina a E/S do processo 1, ele precisa de processador para voltar a executar. Como ele está ocupado com o processo 2, o processo 1 deverá esperar. Por alguma razão, o sistema operacional pode decidir executar o processo 1 imediatamente. Entretanto, o problema não muda. Agora é o processo 2 quem deverá esperar até que o processador fique livre.

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 2.3 ilustra essa situação. O processo 1 ocupa o processador, enquanto os processos 2 e 3 esperam na fila de aptos. Os processos na fila do processador estão no **estado apto** (*ready*). Um único processo ocupa o processador a cada instante. O processo que ocupa o processador está no **estado executando** (*running*). Na figura, o processo 1 está no estado executando, enquanto os processos 2 e 3 estão no estado apto.

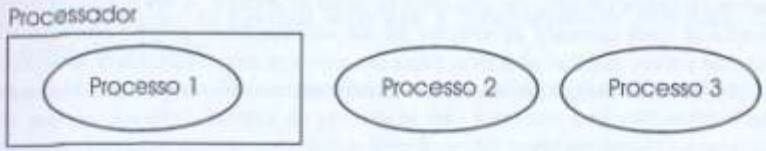

Figura 2.3 - Fila de processos esperando pelo processador.

A Figura 2.4 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*).

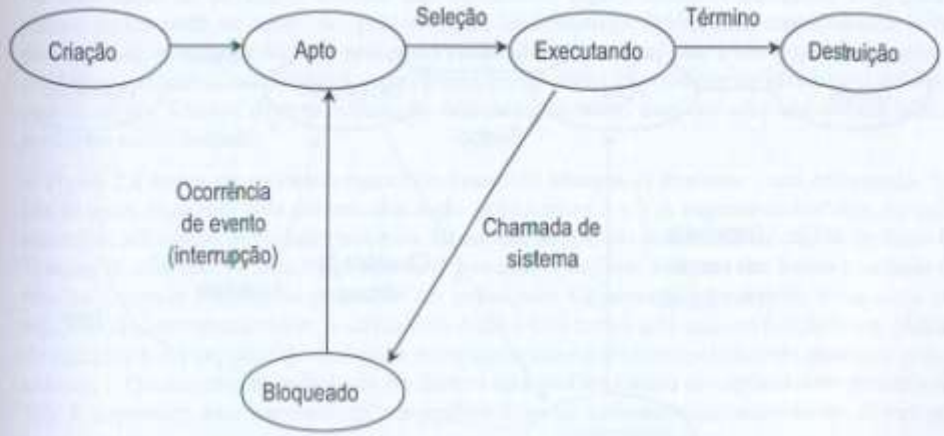

Figura 2.4 - Diagrama de estados de um processo.

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. Como visto antes, 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 de **escalonador** (*scheduler*). Algoritmos para realizar a seleção do processo serão vistos no capítulo sobre gerência do processador.

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. Isso implica um caminho do estado bloqueado para o estado executando.

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. Dessa forma, cada processo tem a chance de executar um pouco. Esse mecanismo cria um caminho entre o estado executando e o estado apto. A Figura 2.5 mostra o grafo de estados dos processos com esses novos caminhos.

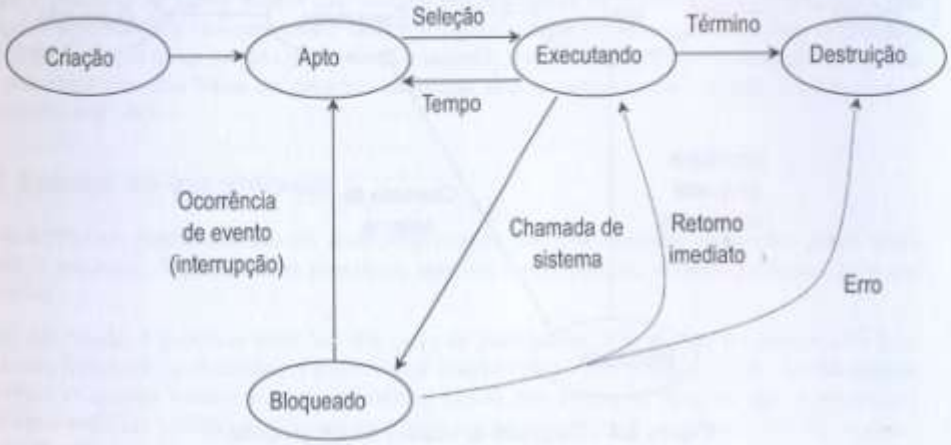

Figura 2.5 – Novo diagrama de estados de um processo

2.6 Gerência de filas

Até aqui, nos exemplos apresentados, sempre um primeiro processo faz uma chamada de sistema e deve esperar. Em seu lugar, um segundo processo recebe o processador. A operação de E/S do primeiro processo termina durante a execução do segundo processo. Entretanto, é preciso lembrar que o tempo de acesso a um periférico é muito grande se comparado aos tempos de processador. Embora a situação apresentada seja possível, é mais provável que o segundo processo também solicite E/S, antes da conclusão do primeiro pedido.

Se a nova solicitação for para outro periférico, o mesmo procedimento será repetido. O comando é enviado para o periférico. O processo solicitante é temporariamente bloqueado. Um terceiro processo recebe o processador.

Se a nova solicitação for para o mesmo periférico, não é possível enviar o comando imediatamente. A grande maioria dos controladores de periféricos não suportam dois comandos simultâneos. O segundo processo terá que esperar o periférico ficar livre. De qualquer maneira, o processador ficou livre. Um terceiro processo será retirado da fila de aptos e passará a ocupar o processador.

No caso de um periférico popular, freqüentemente usado, como o principal disco do computador, essa mesma situação pode ocorrer várias vezes. Por exemplo, o terceiro processo também solicita acesso ao mesmo periférico. Temos, nesse caso, três processos bloqueados, associados ao mesmo periférico. Um deles está acessando, enquanto os outros dois esperam para acessar. Será necessário manter uma fila associada a cada periférico.

Quando uma solicitação é feita, é preciso verificar a fila do periférico. Se a fila estiver vazia, o periférico está livre. O pedido é inserido na fila, e o comando é enviado para o controlador. Caso contrário, o periférico está ocupado. O pedido é inserido na fila, mas nada é enviado ao controlador. Na interrupção do periférico, também são necessários alguns cuidados adicionais. O primeiro pedido da fila pode ser removido, pois o acesso foi concluído. O processo correspondente volta para a fila de aptos, para disputar processador. Se, após a sua remoção, a fila ficou vazia, então o periférico está livre. Caso contrário, é necessário enviar para o controlador do periférico o primeiro pedido da fila. Trata-se de uma solicitação feita anteriormente, mas que não fora enviada pois o periférico estava ocupado.

A Figura 2.6 ilustra um momento específico dentro do sistema. O processo 1 está executando. Na fila de aptos, esperando pelo processador, estão os processos 2 e 3. A impressora está livre, ou seja, não existe solicitação de nenhum processo. Existe um acesso em andamento na unidade de disco 0. Trata-se de uma solicitação do processo 4. O processo 5 também solicitou um acesso à unidade 0, mas deve esperar a conclusão do pedido em andamento. Os processos 4 e 5 estão bloqueados, ou seja, não disputam processador. A unidade de disco 1 está sendo acessada em função de um pedido do processo 6. Ao contrário da unidade 0, nesse momento não existem solicitações pendentes para a unidade 1. O tratamento do acesso aos periféricos será melhor tratado no capítulo sobre gerência de E/S. É importante notar que cada tipo de periférico possui características próprias, que devem ser consideradas.

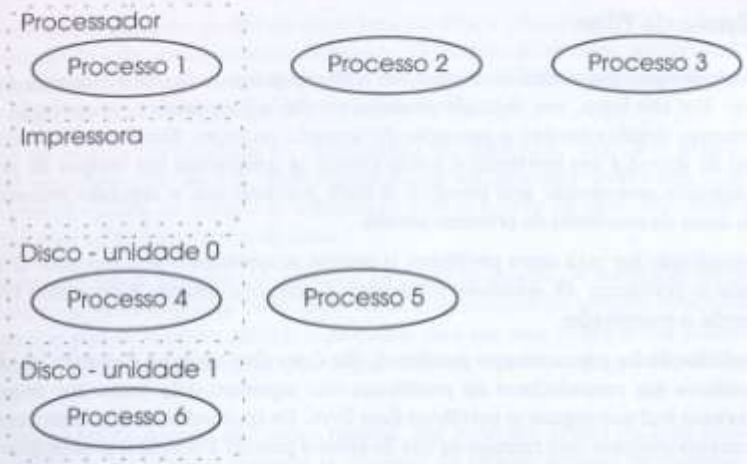

Figura 2.6 - Filas do sistema operacional

2.7 Mecanismo de interrupções

O **mecanismo de interrupções** é um recurso comum dos processadores de qualquer porte. Ele permite que um controlador de periférico chame a atenção do processador. Fisicamente, o barramento de controle é usado para o envio de sinais elétricos associados com a geração de uma interrupção.

É possível fazer uma analogia entre as interrupções de hardware e o telefone. Nesse caso, o processador corresponde a uma pessoa trabalhando (por exemplo, escrevendo um livro de Sistemas Operacionais). Quando o telefone toca, ele sinaliza a ocorrência de uma interrupção para o processador. Imediatamente, a pessoa pára o que está fazendo e vai até o telefone atender a interrupção. Quando o atendimento tiver terminado, a pessoa retorna para a sua tarefa original, no ponto onde parou.

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. Quando o tratador termina, a execução volta para a rotina interrompida, sem que essa perceba que foi interrompida.

Em certos aspectos, uma interrupção é semelhante a uma chamada de sub-rotina. Nos dois casos existe uma rotina que é ativada e, quando termina, a execução retorna para a rotina original. Entretanto, como será visto mais adiante, interrupções tanto podem ser ativadas por software como por hardware. No caso de ativação por hardware, o momento exato não pode ser previsto pelo programa. A princípio, o tratador poderá ser ativado em qualquer ponto da execução de um programa.

Para que a execução do programa interrompido não seja comprometida pela interrupção, é necessário que nenhum registrador seja alterado. Em outras palavras, quando a execução voltar para

o programa interrompido, o conteúdo de todos os registradores deverá ser o mesmo que no momento em que ocorreu a interrupção. Alguns processadores salvam automaticamente todos os registradores quando ocorre uma interrupção. Outros processadores salvam apenas alguns, cabendo à rotina que atende à interrupção salvar os demais registradores. O local mais indicado é a pilha de execução. É usual o processador dispor de uma instrução tipo **retorno de interrupção**. Essa instrução repõe o conteúdo original dos registradores e faz o processador retomar a execução do programa interrompido.

Todo computador possui uma variada gama de periféricos. A função básica de um **controlador de periférico** é conectar o dispositivo em questão com o processador. Através do barramento, o processador é capaz de realizar operações do tipo "lê dados", "escreve dados", "reinicializa", "lê status" e "escreve comando". O controlador é responsável por traduzir essas operações em uma seqüência de acionamentos eletrônicos, elétricos e mecânicos capazes de realizar a operação solicitada. Para isso, o controlador deve saber como o periférico funciona. Isso resulta em cada tipo de periférico necessitar um controlador diferente.

A Figura 2.7 mostra o diagrama de tempo resultante quando interrupções são empregadas para implementar E/S. Após enviar o comando para o controlador, o processador segue a execução do programa. Quando a operação é concluída, o controlador gera uma interrupção. O processador então lê os resultados do controlador.

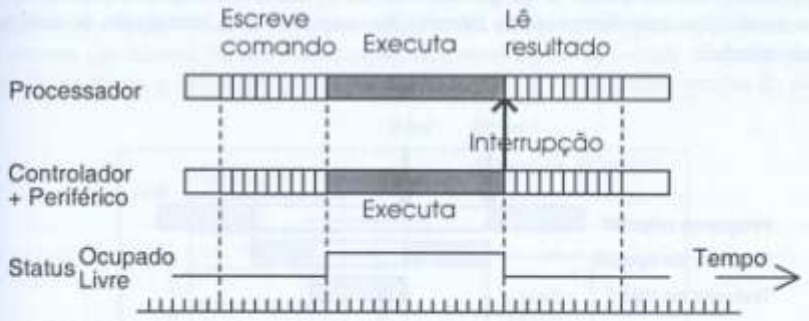

Figura 2.7 - Linha de tempo quando interrupção é usada para implementar E/S.

Em um computador podem existir diversos controladores capazes de gerar interrupções. A forma mais simples de identificar a origem de uma interrupção é associar a cada controlador um tipo diferente de interrupção. Dessa forma, o controlador não somente interrompe o processador, mas também informa qual o tipo da interrupção.

A maioria dos processadores admitem diversos **tipos de interrupções**. Cada tipo de interrupção é identificado por um número. Por exemplo, de 0 (zero) até 255. O significado específico de cada tipo é definido pelos projetistas do sistema. Normalmente, periféricos diferentes geram interrupções de tipos diferentes. Cada tipo de interrupção existente no sistema pode estar associado a um tratador diferente. Também é possível usar a mesma rotina para tratar diferentes tipos de interrupções.

Existem momentos em que um programa não pode ser interrompido. Por exemplo, o programa pode estar alterando variáveis que também são acessadas pelo tratador de interrupções. Durante a alteração, o programa pode deixar essas variáveis temporariamente com valores inconsistentes. Se a

interrupção ocorrer nesse instante, o tratador será ativado e irá acessar essas variáveis, que estão com valores incorretos. É preciso ter em mente que o instante exato em que vai ocorrer uma interrupção de hardware é imprevisível.

A solução para esse problema é desligar o mecanismo de interrupções temporariamente, enquanto o programa realiza uma tarefa crítica que não pode ser interrompida. Os processadores normalmente possuem instruções para **habilitar e desabilitar interrupções**. Enquanto as interrupções estiverem desabilitadas, elas serão ignoradas pelo processador. Elas não são perdidas, apenas ficam pendentes. Quando o programa tornar a habilitar as interrupções, elas imediatamente serão atendidas pelo processador. Com as interrupções desabilitadas, o acesso às estruturas de dados pode ser feito de forma segura.

É comum a existência de uma relação de **prioridades** entre os diferentes tipos de interrupções. Vamos supor que um dado processador atribui prioridade decrescente do tipo 0 até o tipo 255. Nesse caso, quando ocorre uma interrupção tipo 10, o processador automaticamente desabilita interrupções dos tipos 10 a 255 até que o atendimento da interrupção tipo 10 tenha sido concluído. Caso ocorra, durante esse intervalo de tempo, uma interrupção do tipo 5, ela é imediatamente reconhecida pelo processador. Assim, o tratador de interrupção tipo 10 é ele próprio interrompido e o tratador de interrupções tipo 5 ativado. Quando esse termina, o tratador de interrupções tipo 10 é retomado. Por sua vez, quando esse termina, o programa que estava executando originalmente é retomado. A Figura 2.8 ilustra essa situação. Quando tratadores de interrupção podem, eles próprios, serem interrompidos, a programação torna-se complexa. Em função disso, muitos sistemas desabilitam completamente as interrupções enquanto uma interrupção de qualquer tipo está sendo atendida.

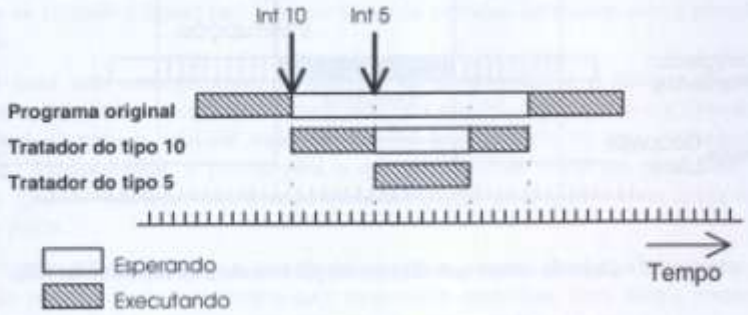

Figura 2.8 - Ativação em cascata de tratadores de interrupção.

O endereço de um tratador de interrupção é chamado de **vetor de interrupção**. O termo é usado pois ele "aponta" para a rotina de atendimento da interrupção. Cada tipo de interrupção possui associado um vetor de interrupção. Em geral, existe uma tabela na memória com todos os vetores de interrupção, ou seja, com os endereços das rotinas responsáveis por tratar os respectivos tipos de interrupção. A Figura 2.9 ilustra o conceito de **tabela dos vetores de interrupção**. Para que uma mesma rotina atenda diversos tipos de interrupção, basta colocar o seu endereço em diversas entradas da tabela. Entretanto, o mais comum é utilizar uma rotina para cada tipo de interrupção.

A ocorrência de uma interrupção dispara no processador uma **seqüência de atendimento**. O processador verifica se o tipo de interrupção sinalizado está habilitado. Caso negativo, é ignorado. Se estiver habilitado, o conteúdo dos registradores é salvo na pilha, o endereço da rotina responsável pelo tratamento daquele tipo de interrupções é obtido na entrada correspondente ao número da interrupção que ocorreu e a execução é desviada para esse endereço. Toda essa seqüência é executada pelo hardware, comandada pela unidade de controle do processador.

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 é quem 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 conhecer o tipo de interrupção apropriado.

O maior uso para interrupções de software é a implementação das **chamadas de sistema**, através das quais os programas de usuários solicitam serviços ao sistema operacional. Os endereços das rotinas de um sistema operacional mudam, conforme a versão do sistema utilizado. Se os programas dependessem desse endereço para chamar o sistema operacional, eles teriam que ser recompilados a cada nova versão do sistema. Em vez de chamar as rotinas do sistema operacional através dos seus endereços, interrupções de software são utilizadas. Embora o endereço da rotina seja alterado de versão para versão, o número (tipo) da interrupção de software usado para ativá-la permanece sempre o mesmo. O endereço armazenado na tabela de vetores de interrupção deve mudar conforme a versão. Mas os programas de usuários não precisam preocupar-se com isso. O próprio sistema operacional, na sua inicialização, fica encarregado de colocar os endereços certos na tabela. Dessa forma, o mesmo programa é capaz de funcionar em diferentes versões do sistema.

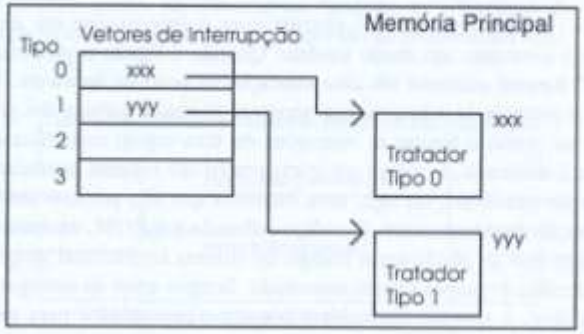

Figura 2.9 - Tabela dos vetores de interrupção.

Não é possível desabilitar interrupções de software, mesmo porque não é necessário. Somente quem pode gerar uma interrupção de software é a rotina em execução. Se a rotina em execução não deseja que interrupções de software aconteçam, basta não gerar nenhuma. Entretanto, a ocorrência de uma interrupção de software desabilita interrupções exatamente da mesma forma que acontece com as interrupções de hardware. Na maioria dos processadores, a mesma seqüência de eventos que ocorrem após uma interrupção de hardware acontece também após uma interrupção de software.

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 na verdade não existe. O procedimento de atendimento a essa classe de interrupções é igual ao descrito anteriormente.

2.8 Proteção entre processos

Na multiprogramação diversos processos compartilham o computador. É necessário que o sistema operacional ofereça proteção aos processos e garanta a utilização correta do sistema. Por exemplo, um processo de usuário não pode formatar o disco. Também não se pode permitir que um processo entre em um laço infinito e, com isso, monopolize a utilização do processador. Nessa seção, serão descritos mecanismos cuja função é garantir a correta operação do sistema.

2.8.1 Modos de operação do processador

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** 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**, e somente podem ser executadas em modo supervisor. 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. O mesmo acontece em uma operação de *reset* do hardware. No *reset*, o sistema operacional recebe o controle da máquina. Isso acontece porque a maioria dos processadores, ao ser ligado ou sofrer *reset*, passa a buscar as instruções de uma região específica da memória. Nessa região de memória é colocado o código de inicialização do sistema operacional. Esse código é normalmente colocado em ROM, ou seja, uma memória que não perde o seu conteúdo quando é desligada a alimentação do computador. O código colocado em ROM, na maioria das vezes, busca no disco ou em outro tipo de periférico o código do sistema operacional propriamente dito. Ele é carregado para a memória principal e então executado. Sempre antes de entregar o processador para um processo de usuário, o sistema operacional comuta o processador para modo usuário. Dessa forma, processos de usuário executam em modo usuário.

2.8.2 Proteção dos periféricos

Para proteger os periféricos, as instruções de E/S são tornadas privilegiadas. Se um processo de usuário tentar acessar diretamente um periférico, ocorre uma interrupção. O sistema operacional é ativado, já em modo supervisor, e o processo de usuário é abortado, pois tentou um acesso ilegal. A única forma de o processo de usuário realizar uma operação de E/S é através de uma chamada de sistema.

Quando ocorre uma interrupção do periférico, o sistema operacional é ativado. Como ocorreu uma interrupção, o processador é chaveado para modo supervisor. Por exemplo, para atender as

chamadas de sistema, ele precisa acessar os periféricos. Como foi dito antes, isso agora só é possível com o processador em modo supervisor.

Muitas arquiteturas oferecem uma instrução chamada de **interrupção de software** ou ***trap***. Essa instrução, quando executada, apresenta um efeito semelhante ao de uma interrupção por hardware. A única diferença está no fato de ter sido causada pelo software e não por um evento externo vinculado ao hardware. As interrupções geradas por software também chaveiam o processador para modo supervisor. Normalmente as interrupções de software são utilizadas para implementar as chamadas de sistema, pois permitem automaticamente a passagem ao modo supervisor.

A Figura 2.10 ilustra as situações em que ocorrem trocas do modo de operação. Observe que o sistema operacional é ativado em três situações. Primeiro, por uma interrupção de periférico, informando a conclusão de alguma operação de E/S. Segundo, por uma interrupção do hardware de proteção, porque o processo em execução tentou uma operação ilegal. Esse hardware de proteção aparece tipicamente incorporado ao próprio processador, fazendo parte do mesmo circuito integrado. Finalmente, por uma interrupção de software, que representa uma chamada de sistema do processo em execução. Em todas elas, o sistema operacional será ativado em modo supervisor.

2.8.3 Proteção da memória

Pelo que foi apresentado até aqui, é possível perceber que a proteção das rotinas que atendem interrupções é vital. Se o usuário conseguir instalar suas rotinas no lugar dos tratadores de interrupção do sistema operacional, ele conseguirá o processador em modo supervisor. Além disso, um usuário poderá corromper o sistema se for capaz de alterar áreas de código ou variáveis do sistema operacional. É necessário proteger a memória do sistema operacional, tanto código como dados. Ao mesmo tempo, é necessário proteger a memória usada por um processo do acesso de outros processos. Deve ser impossível para o processo do usuário João alterar a memória alocada para o processo da usuária Maria. Se isso não for verdade, os programas da Maria poderão não funcionar em decorrência de um comportamento indevido do processo do João.

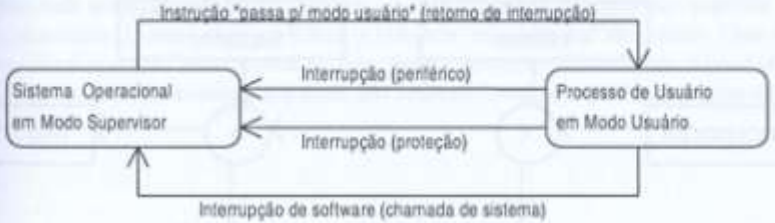

Figura 2.10 - Modos de operação do processador.

Para implementar a proteção de memória, o sistema operacional também necessita de auxílio da arquitetura. Existem muitas maneiras de realizar a proteção da memória, sendo as mais importantes apresentadas no capítulo sobre gerência de memória. Nessa seção é apresentada uma forma clássica ([DEI84], [SIL85]), simples mas eficaz. O propósito é mostrar que tal proteção é perfeitamente possível, a partir de um suporte apropriado do hardware. Serão empregados **registradores de limite**. A Figura 2.11 ilustra a sua utilização. Sempre que o sistema operacional vai disparar um processo de usuário, ele carrega nos registradores de limite os valores relativos ao processo que vai

executar. Ele coloca no registrador limite inferior o endereço do primeiro byte pertencente à área de trabalho do processo. No registrador limite superior, é colocado o endereço do último byte válido para a sua área.

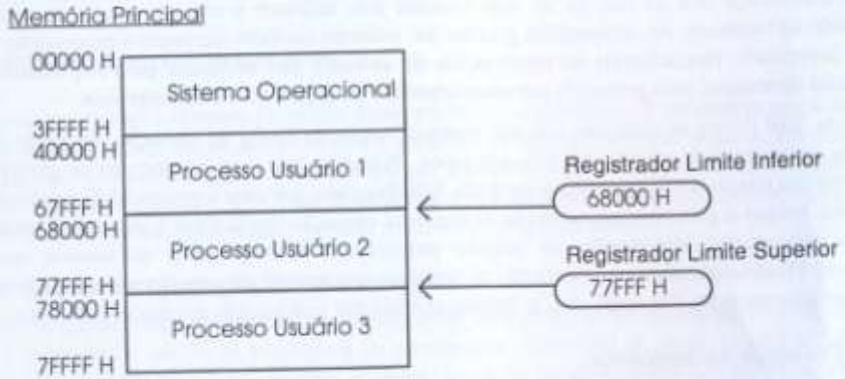

Figura 2.11 - Proteção de memória com registradores de limite.

A cada acesso à memória, o hardware de proteção compara o endereço gerado pelo processador com o conteúdo dos dois registradores de limite. Se o endereço gerado estiver fora da área do usuário, é gerada uma interrupção. Nesse caso, o sistema operacional é ativado em modo supervisor. O processo de usuário será abortado, devido a um acesso ilegal à memória. A Figura 2.12 ilustra esse hardware de proteção.

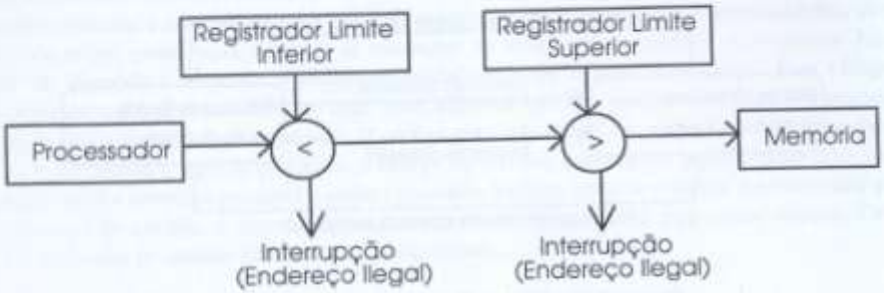

Figura 2.12 - Hardware para verificação dos endereços.

Algumas arquiteturas fazem o acesso aos periféricos através de posições específicas da memória. Essa técnica é chamada de **E/S mapeada na memória**. Nesse caso, a proteção de memória também implementa a proteção dos periféricos. Basta colocar os endereços dos periféricos fora dos limites da área do processo. O processo não será capaz de acessar o periférico sem gerar uma interrupção.

Obviamente, o acesso aos registradores de limite deve ser feito através de instruções privilegiadas. Somente o sistema operacional pode alterar o seu conteúdo. Por outro lado, o sistema operacional

deve acessar a área do usuário. Isso é necessário para o sistema operacional colocar lá o próprio programa do usuário. Muitas chamadas de sistema também fazem com que o sistema operacional busque parâmetros ou coloque respostas diretamente na área do processo usuário. Por essas razões, em modo supervisor, o hardware de proteção pode ser desativado. Ou seja, o hardware de proteção somente gera interrupções quando o processador está em modo usuário.

Para evitar que um único processo de usuário monopolize a utilização do processador, é empregado um **temporizador** (*timer*). O temporizador é um **relógio de tempo real**, implementado pelo hardware, que gera interrupções de tempos em tempos. O período do temporizador corresponde ao intervalo de tempo entre interrupções. Em geral, ele é da ordem de milisegundos. As interrupções do temporizador ativam o sistema operacional. Ele então verifica se o processo executando já está há muito tempo com o processador. Nesse caso, o sistema pode abortar o processo por exceder o limite de tempo de execução. Essa é uma prática usual em sistemas nos quais a execução não é acompanhada pelo usuário, e um laço infinito no programa não seria percebido. Em outros sistemas é usual o processo voltar para o fim da fila de aptos. Com isso, todos os processos teriam chance de executar. Mais tarde, o processo interrompido teria oportunidade para executar novamente.

O controle do temporizador deve ser feito através de instruções privilegiadas. Se o processo usuário desligar o temporizador, ele terá controle do processador por tempo indeterminado. O temporizador ligado é a garantia de que o sistema operacional será acionado ao menos uma vez a cada período de alguns milisegundos.

Todo o mecanismo de proteção está baseado em interrupções. O processo usuário deve sempre executar com as interrupções habilitadas. Somente assim, uma operação ilegal será detectada. Ao mesmo tempo, a instrução que desabilita interrupções deve ser privilegiada. Isso é necessário para que o processo usuário não possa desabilitar as interrupções e tornar o mecanismo inoperante.

Os mecanismos de proteção apresentados nesse capítulo não são os únicos. Por exemplo, a forma de proteger a memória depende muito do mecanismo de gerência de memória empregado. O objetivo dessa seção foi mostrar a necessidade de proteção e o princípio básico da operação dual do processador. Na verdade, muitas arquiteturas não definem apenas dois modos de operação, mas vários. Em cada **modo de operação**, também chamado de **nível de proteção**, algumas operações não são permitidas. Quanto mais confiável o software, mais direitos ele recebe. Com auxílio do hardware, como mostrado neste capítulo, e um sistema operacional sem falhas, é possível construir sistemas seguros, ou seja, sistemas nos quais não ocorram interferências danosas entre usuários.

2.9 Exercícios

1) O sistema operacional é um programa dirigido por eventos, e esses eventos são sinalizados por interrupções. Para cada uma das três classes de interrupções (periférico, proteção, chamada de sistema), descreva a reação que o sistema operacional deverá ter. Em outras palavras, o que o sistema operacional deverá fazer em função do evento sinalizado. (Seções 2.5/2.8)

2) Os três principais estados de um processo são:

- Apto a executar (*ready*);

- Executando (*running*);

- Esperando pela entrada/saída (*blocked*).

Descreva os eventos que fazem com que um processo mude de estado. (Seção 2.5)

3) A operação "passa para modo usuário" deve ou não ser privilegiada? Justifique. (Seção 2.8)

4) A operação "desabilita interrupções" deve ou não ser privilegiada? Justifique. (Seção 2.8)

5) A operação "escreve caractere na interface da impressora" deve ou não ser privilegiada? Justifique. (Seção 2.8)

6) A operação "desliga o temporizador" deve ou não ser privilegiada? Justifique. (Seção 2.8)

7) Pode-se considerar como conseqüência da multiprogramação "uma pior utilização do processador"? Justifique. (Seção 2.1)

8) Pode-se considerar como conseqüência da multiprogramação "uma pior utilização dos periféricos"? Justifique. (Seção 2.1)

9) Pode-se considerar como conseqüência da multiprogramação "uma menor necessidade de memória"? Justifique. (Seção2.1)

10) Pode-se considerar como conseqüência da multiprogramação "uma menor necessidade de hardware para proteção"? Justifique. (Seção 2.1)

11) Muitas arquiteturas dividem as instruções em normais e privilegiadas (restritas). Mostre como isso pode ser utilizado para impedir que processos tenham acesso direto aos periféricos, mas ainda possam fazer as operações necessárias de entrada e saída (descreva o mecanismo). (Seção 2.8)

12) Explique como o mecanismo de modos de execução do processador, associado com o mecanismo de interrupções, pode impedir que um processo executando código de usuário possa, por exemplo, acessar diretamente o controlador do disco. (Seção 2.8)

13) Explique por que é vantajoso associar a passagem do processador de modo usuário para modo supervisor com o atendimento de uma interrupção. (Seção 2.8)

14) Explique em que situações (que tipos de interrupções) ocorre a passagem do processador de modo usuário para modo supervisor. (Seção 2.8)

3
Programação Concorrente

Programação concorrente tem sido usada freqüentemente na construção de sistemas operacionais e em aplicações nas áreas de comunicação de dados e controle industrial. Esse tipo de programação torna-se ainda mais importante com o advento dos sistemas distribuídos e das máquinas com arquitetura paralela. Neste capítulo serão discutidos os conceitos básicos e alguns mecanismos clássicos da programação concorrente. Maiores detalhes podem ser encontrados no livro [TOS03].

3.1 Definição

Um programa que é executado por apenas um processo é chamado de **programa seqüencial**. A grande maioria dos programas escritos são programas seqüenciais. Nesse caso, existe somente um fluxo de controle durante a execução. Isso permite, por exemplo, que o programador realize uma "execução imaginária" de seu programa apontando com o dedo, a cada instante, a linha do programa que está sendo executada no momento.

Um **programa concorrente** é executado simultaneamente por diversos processos que cooperam entre si, isto é, trocam informações. Para o programador realizar agora uma "execução imaginária", ele vai necessitar de vários dedos, um para cada processo que faz parte do programa. Nesse contexto, trocar informações significa trocar dados ou realizar algum tipo de sincronização. É necessária a existência de interação entre processos para que o programa seja considerado concorrente. Embora a interação entre processos possa ocorrer através do acesso a arquivos comuns, esse tipo de concorrência é tratada na disciplina de Banco de Dados. A programação concorrente tratada neste livro, assim como nas disciplinas de Sistemas Operacionais, utiliza mecanismos rápidos para interação entre processos: variáveis compartilhadas e troca de mensagens.

O termo "programação concorrente" vem do inglês *concurrent programming*, onde *concurrent* significa "acontecendo ao mesmo tempo". Uma tradução mais exata seria programação concomitante. Entretanto, o termo programação concorrente já está solidamente estabelecido no Brasil. Algumas vezes é usado o termo **programação paralela** com o mesmo sentido.

O verbo "concorrer" admite em português vários sentidos. Pode ser usado no sentido de cooperar, como em "tudo concorria para o bom êxito da operação". Também pode ser usado com o significado de disputa ou competição, como em "ele concorreu a uma vaga na universidade". Em uma forma menos comum ele significa também existir simultaneamente. De certa forma, todos os sentidos são aplicáveis aqui na programação concorrente. Em geral, processos concorrem (disputam) pelos mesmos recursos do hardware e do sistema operacional. Por exemplo, processador, memória, periféricos, estruturas de dados, etc. Ao mesmo tempo, pela própria

definição de programa concorrente, eles concorrem (cooperam) para o êxito do programa como um todo. Certamente, vários processos concorrem (existem simultaneamente) em um programa concorrente. Logo, programação concorrente é um bom nome para o que vamos tratar neste capítulo.

É comum em sistemas multiusuário que um mesmo programa seja executado simultaneamente por vários usuários. Por exemplo, um editor de texto. Entretanto, executar simultaneamente 10 instâncias do editor de texto não faz dele um programa concorrente. Apenas o código é possivelmente compartilhado pelos 10 processos. Cada processo executa sobre sua própria área de dados e ignora a existência de outras execuções do programa. Esses processos não cooperam entre si, isto é, não trocam informações. Nesse exemplo, temos apenas a execução de 10 instâncias do mesmo programa seqüencial, e não um programa concorrente.

3.2 Motivação

Notadamente, a programação concorrente é mais complexa que a programação seqüencial. Um programa concorrente pode apresentar todos os tipos de erros que normalmente aparecem em programas seqüenciais. Além disso, existem os erros associados com as interações entre os processos. Muitos erros dependem da velocidade relativa dos processos. Ou ainda, do exato instante de tempo em que o escalonador do sistema operacional realizou um chaveamento de contexto. Isso torna muitos erros difíceis de reproduzir e de identificar.

Mesmo com todas as suas complexidades inerentes, existem muitas áreas nas quais a programação concorrente é útil. Em sistemas nos quais existem vários processadores (máquinas paralelas ou sistemas distribuídos), é possível aproveitar esse paralelismo explicitamente e acelerar a execução do programa. Mesmo em sistemas com um único processador, existem razões para o seu uso em determinados tipos de aplicações.

Considere um programa que deve ler registros de um arquivo, colocar em um formato apropriado e então enviar para uma impressora física (em oposição a uma impressora lógica ou virtual, implementada com arquivos). Podemos fazer isso com um programa seqüencial que, dentro de um laço, faz as três operações. A Figura 3.1 ilustra tal programa, e a Figura 3.2 mostra a respectiva linha de tempo.

Figura 3.1 - Programa seqüencial acessando arquivo e impressora.

Inicialmente o processo envia um comando para a leitura do arquivo e fica bloqueado. O disco então é acionado para realizar a operação de leitura. Uma vez concluída a leitura, o processo realiza a formatação e inicia a transferência dos dados para a impressora. Como trata-se de uma impressora física, o processo executa um laço no qual os dados são enviados para a porta serial ou paralela apropriada. Como o *buffer* da impressora é relativamente pequeno, o processo fica preso até o final da impressão. Observe no diagrama da Figura 3.2 que o disco e a impressora nunca trabalham

simultaneamente, embora não exista nenhuma limitação de natureza eletrônica. O programa seqüencial é que não consegue ocupar ambos.

Figura 3.2 - Linha de tempo do programa seqüencial da Figura 3.1.

Vamos agora empregar um programa concorrente como o mostrado na Figura 3.3 para realizar a impressão do arquivo. Dois processos dividem o trabalho. O processo leitor é responsável por ler registros do arquivo, formatar e colocar em um *buffer* na memória. O processo impressor retira os dados do *buffer* e envia para a impressora. É suposto aqui que os dois processos possuem acesso à memória onde está o *buffer*. A Figura 3.4 mostra a linha de tempo resultante.

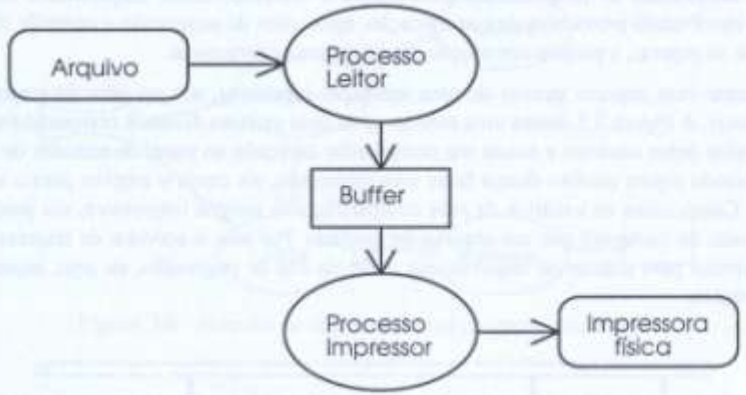

Figura 3.3 - Programa concorrente acessando arquivo e impressora.

O programa concorrente é mais eficiente, pois consegue manter o disco e a impressora trabalhando simultaneamente. O tempo total para realizar a impressão do arquivo é menor quando a solução concorrente é empregada. É claro que a solução possui limitações. Se o processo leitor for sempre mais rápido, o *buffer* ficará cheio, e então o processo leitor terá que esperar até que o processo impressor retire algo do *buffer*. Por outro lado, se o processo impressor for sempre mais rápido, eventualmente o *buffer* ficará vazio e ele terá que esperar pelo processo leitor. É isso que acontece

no diagrama da Figura 3.4. De qualquer forma, a solução concorrente para esse problema nunca será mais lenta que a solução seqüencial.

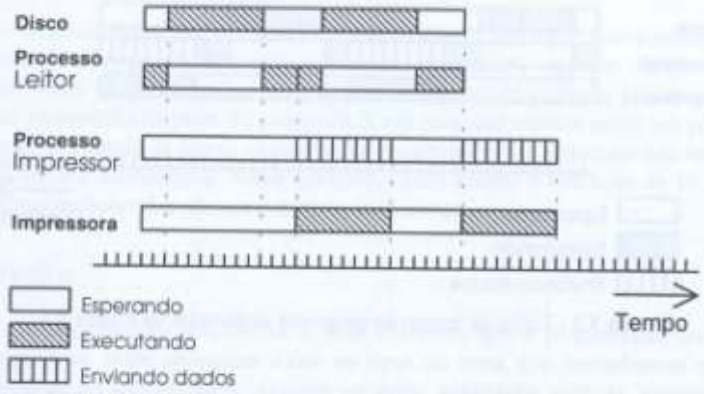

Figura 3.4 - Linha de tempo do programa concorrente da Figura 3.3.

Na verdade, a maior motivação para a programação concorrente é a engenharia de software. Aplicações inerentemente paralelas (aplicações que possuem paralelismo intrínseco) são mais facilmente construídas se programação concorrente é utilizada. Estão enquadrados nesse grupo aplicações envolvendo protocolos de comunicação, aplicações de supervisão e controle industrial e, como era de se esperar, a própria construção de um sistema operacional.

Vamos ilustrar esse aspecto através de uma aplicação hipotética, um servidor de impressão para uma rede local. A Figura 3.5 ilustra uma rede local na qual existem diversos computadores pessoais (PC) utilizados pelos usuários e existe um computador dedicado ao papel de servidor de impressão da rede. Quando algum usuário deseja fazer uma impressão, ele envia o arquivo para o servidor de impressão. Como todos os usuários da rede compartilham a mesma impressora, ela possivelmente estará ocupada no momento que um arquivo for enviado. Por isso, o servidor de impressão usa um disco magnético para manter os arquivos que estão na fila de impressão, ou seja, esperando para serem impressos.

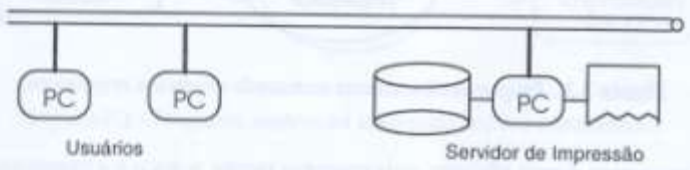

Figura 3.5 - Rede local incluindo um servidor de impressão dedicado.

É importante observar que o programa "servidor de impressão" possui um certo grau de paralelismo intrínseco. Ele deve: receber mensagens pela rede; escrever em disco os pedaços de arquivos recebidos; enviar mensagens pela rede contendo, por exemplo, respostas às consultas sobre o seu estado; ler arquivos previamente recebidos; enviar dados para a impressora. Todas essas atividades devem ser realizadas simultaneamente. Uma forma clara de programarmos o servidor de impressão é usar vários processos de tal forma que cada processo fique responsável por uma atividade em particular. Obviamente, esses processos vão precisar trocar informações para realizar o seu trabalho. Logo, teremos o servidor de impressão implementado na forma de um programa concorrente.

A Figura 3.6 mostra uma das possíveis soluções para a organização interna do programa concorrente "servidor de impressão". Cada círculo representa um processo. Cada flecha representa a passagem de dados de um processo para o outro. Essa passagem de dados pode ser feita, por exemplo, através de variáveis que são compartilhadas pelos processos envolvidos na comunicação.

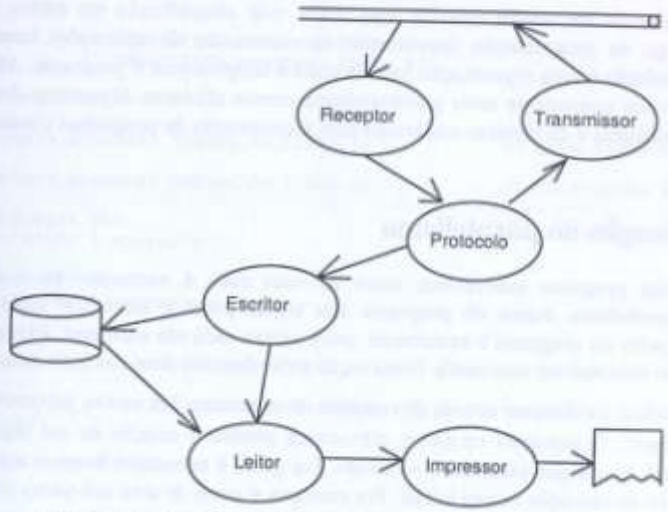

Figura 3.6 - Servidor de impressão como programa concorrente.

Vamos agora descrever rapidamente a função de cada processo. O processo "Receptor" é responsável por receber mensagens da rede local. Ele faz isso através de chamadas de sistema apropriadas e descarta as mensagens com erro. As mensagens corretas são então passadas para o processo "Protocolo". Ele analisa o conteúdo das mensagens recebidas à luz do protocolo de comunicação suportado pelo servidor de impressão. É possível que seja necessário a geração e o envio de mensagens de resposta. O processo "Protocolo" gera as mensagens a serem enviadas e passa-as para o processo "Transmissor", que as envia através de chamadas de sistema apropriadas.

Algumas mensagens contêm pedaços de arquivos a serem impressos. É suposto aqui que mensagens são da ordem de alguns Kbytes. Dessa forma, um arquivo deve ser dividido em várias mensagens para transmissão através da rede. Quando o processo "Protocolo" identifica uma mensagem que contém um pedaço de arquivo, ele passa esse pedaço de arquivo para o processo "Escritor". Passa também a identificação do arquivo ao qual o pedaço em questão pertence. Cabe ao processo "Escritor" usar as chamadas de sistema apropriadas para escrever no disco. Quando o pedaço de arquivo em questão é o último de seu arquivo, o processo "Escritor" passa para o processo "Leitor" o nome do arquivo, que está pronto para ser impresso.

O processo "Leitor" executa um laço externo no qual ele pega um nome de arquivo, envia o conteúdo para o processo "Impressor" e então remove o arquivo lido. O envio do conteúdo para o processo "Impressor" é feito através de um laço interno composto pela leitura de uma parte do arquivo e pelo envio dessa parte. Finalmente, o processo "Impressor" é encarregado de enviar os pedaços de arquivo que ele recebe para a impressora. O relacionamento entre os processos "Leitor" e "Impressor" foi descrito antes, no início desta seção.

Embora o servidor de impressão descrito aqui tenha sido simplificado em vários aspectos, ele ilustra o emprego da programação concorrente na construção de aplicações com paralelismo intrínseco. O resultado é uma organização interna clara e simples para o programa. Além disso, um programa seqüencial equivalente seria provavelmente menos eficiente. O restante deste capítulo é dedicado aos problemas e às técnicas existentes para a construção de programas concorrentes como esse.

3.3 Especificação do paralelismo

Para construir um programa concorrente, antes de mais nada é necessário ter a capacidade de **especificar o paralelismo** dentro do programa. Em algum ponto é necessário especificar quantos processos farão parte do programa e exatamente quais rotinas cada um executará. Existem, na prática, diversas maneiras para realizar essa tarefa. Nesta seção serão descritas duas das mais usuais.

É possível especificar paralelismo através do conjunto de comandos: "`create_process`", "`exit`" e "`wait_process`". O comando `create_process` permite a criação de um segundo fluxo de execução, paralelo àquele que executou o comando. Em geral, é necessário fornecer uma indicação de onde o novo fluxo de execução deverá iniciar. Por exemplo, o nome de uma sub-rotina do programa.

Os comandos "`exit`" e "`wait_process`" são auxiliares ao `create_process`. Quando o comando `exit` é executado, o fluxo de controle que o executa é imediatamente terminado. Um comando imediatamente após o comando `exit` jamais será executado. O comando `wait_process` permite que um fluxo de execução espere outro fluxo terminar. Em geral, é necessário fornecer uma indicação do fluxo de execução cujo término está sendo esperado. Por exemplo, um número inteiro retornado pelo comando `create_process`.

Vamos usar a sintaxe da linguagem C para exemplificar o uso dos comandos `create_process`, `exit` e `wait_process`. Considere o trecho de código mostrado na Figura 3.7. Nesse programa, o processo inicial (pai) executa duas vezes o comando `create_process`, criando dois processos adicionais (filhos 1 e 2). Ele então espera que cada um dos filhos termine, escreve uma mensagem para cada um e termina também. Os dois processos criados executam a mesma função "codigo_do_filho". Eles apenas colocam uma mensagem na tela e terminam. Uma possível saída para esse programa é:


```
Alo do pai
Alo do filho
Alo do filho
Filho 1 morreu
Filho 2 morreu
```

A Figura 3.8 mostra um diagrama de tempo capaz de gerar a saída mostrada acima. É importante observar que processos paralelos podem executar em qualquer ordem. Na saída mostrada antes, o segundo filho foi executado antes que o pai percebesse a morte do primeiro filho. Caso o segundo filho tivesse executado após a morte do primeiro filho, a saída seria:

```
Alo do pai
Alo do filho
Filho 1 morreu
Alo do filho
Filho 2 morreu
```

```
/* Programa principal */
main()
{
      int f1;      /* Identifica processo filho 1*/
      int f2;      /* Identifica processo filho 2*/

      printf("Alo do pai\n");

      f1 = create_process( codigo_do_filho );          /* Cria filho 1 */

      f2 = create_process( codigo_do_filho );          /* Cria filho 2 */

      wait_process( f1);
      printf("Filho 1 morreu\n");

      wait_process( f2);
      printf("Filho 2 morreu\n");

      exit();
}

/* Funcao executada pelos dois processos filhos */
codigo_do_filho()
{
      printf("Alo do filho\n");
      exit();
}
```

Figura 3.7 - Exemplo contendo os comandos `create_process`, `exit` e `wait_process`.

A Figura 3.9 mostra o diagrama de tempo associado com essa segunda possibilidade. Essa é uma característica importante dos programas concorrentes. Duas execuções consecutivas do mesmo programa, com os mesmos dados de entrada, podem gerar resultados diferentes. Isso não é necessariamente um erro. Cabe ao programador fazer com que todos os resultados possíveis sejam igualmente corretos. Mais adiante, serão apresentados mecanismos capazes de controlar explicitamente a ordem de execução dos processos, se assim for desejado.

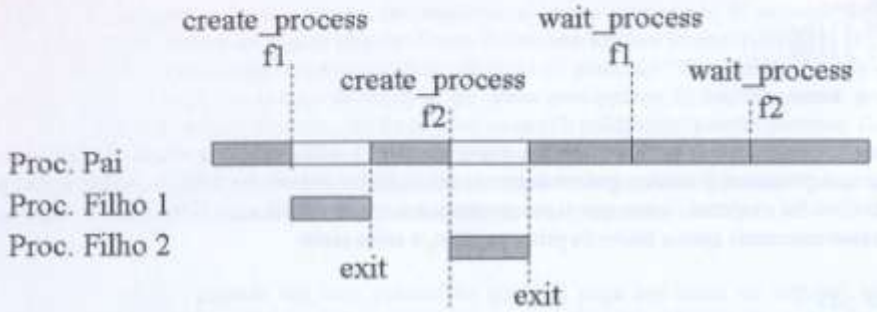

Figura 3.8 - Diagrama de tempo associado com o programa da Figura 3.7.

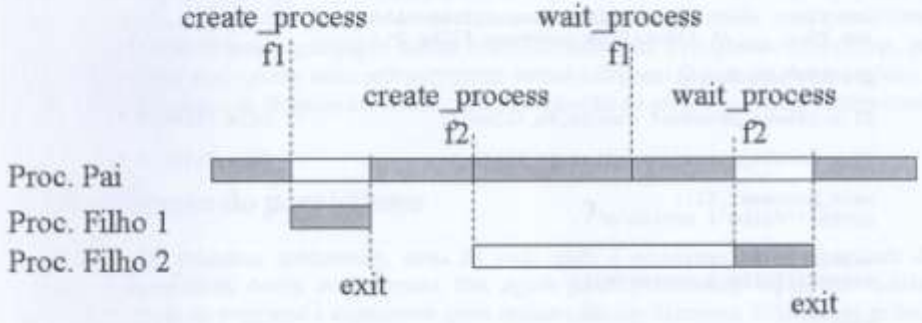

Figura 3.9 - Diagrama de tempo associado com o programa da Figura 3.7.

Muitas vezes são usados grafos de precedência para representar o paralelismo existente em um programa. No grafo de precedência, os nodos representam trechos de código e os arcos representam relações de precedência. Um arco do nodo X para o nodo Y representa que o código associado com Y somente poderá ser executado após o término do código associado com X. Por exemplo, o programa da Figura 3.7 pode ser representado pelo grafo que aparece na Figura 3.10. É importante notar que, nas figuras anteriores, os arcos representavam o fluxo dos dados, enquanto na Figura 3.10 (um grafo de precedência), os arcos não representam fluxo de dados, mas sim o fluxo de controle.

Vamos supor agora que o programador deseje que, necessariamente, o processo do filho 1 termine para então o processo do filho 2 iniciar. A Figura 3.11 mostra como ficaria o código nesse caso.

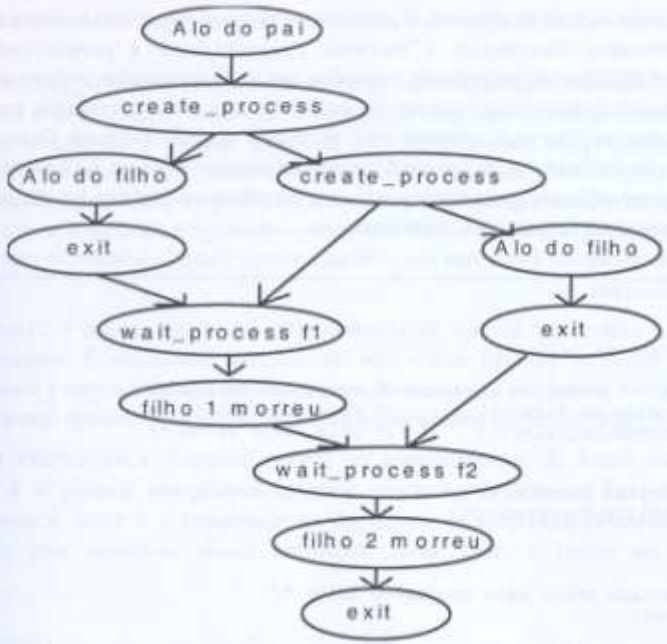

Figura 3.10 – Grafo de precedência para o programa da Figura 3.7.

```
/* Programa principal */
main()
{
      int f1;      /* Identifica processo filho 1*/
      int f2;      /* Identifica processo filho 2*/

      printf("Alo do pai\n");

      f1 = create_process( codigo_do_filho );          /* Cria filho 1 */
      wait_process( f1);
      printf("Filho 1 morreu\n");

      f2 = create_process( codigo_do_filho );          /* Cria filho 2 */
      wait_process( f2);
      printf("Filho 2 morreu\n");

      exit();
}

/* Funcao executada pelos dois processos filho */
codigo_do_filho()
{
      printf("Alo do filho\n");
      exit();
}
```

Figura 3.11 - Programa do exemplo 3.7 alterado.

Uma forma mais estruturada de especificar paralelismo em programas concorrentes é conseguida com o uso dos comandos "Parbegin" e "Parend" (*"parallel begin"* e *"parallel end"*). Enquanto o par Begin-End delimita um conjunto de comandos que serão executados seqüencialmente, o par Parbegin-Parend delimita um conjunto de comandos que serão executados em paralelo. O comando que segue ao Parend somente será executado quando todos os fluxos de controle criados na execução do Parbegin-Parend tiverem terminado. A Figura 3.12 mostra como esses comandos podem ser utilizados para implementar um programa equivalente àquele apresentado na Figura 3.7. Novamente, a linguagem C é utilizada.

```
/* Programa principal */
main()
{
      printf("Alo do pai\n");

      Parbegin     /* Define conjunto de execuções paralelas */
            codigo_do_filho();        /* Cria um filho */
            codigo_do_filho();        /* Cria outro filho */
      Parend;

      printf("Filho 1 morreu\n");
      printf("Filho 2 morreu\n");

}

/* Funcao executada pelos dois processos filho */
codigo_do_filho()
{
      printf("Alo do filho\n");
}
```

Figura 3.12 - Exemplo contendo os comandos Parbegin e Parend.

Os comandos create_process, wait_process e exit são facilmente incorporados à uma linguagem de programação procedural. Nos exemplos mostrados antes, eles foram incorporados à linguagem C na forma de funções de uma biblioteca especial. Já os comandos Parbegin-Parend definem uma nova estrutura de controle para a linguagem. Nos exemplos, a sintaxe da linguagem C foi estendida com a inclusão desse novo comando. Também é possível usar uma versão menos elegante para o comando Parbegin que, entretanto, mantém sua funcionalidade. Dessa vez, Parbegin aparece como uma função de biblioteca cujos parâmetros são os nomes das funções a serem disparadas em paralelo. O exemplo da Figura 3.12 seria adaptado para conter:

```
Parbegin( codigo_do_filho, codigo_do_filho);
```

que substitui a estrutura de controle Parbegin-Parend. De qualquer forma, fica evidente que não é difícil adaptar as linguagens de programação para, de alguma forma, permitir a especificação de paralelismo dentro dos programas.

Vamos agora considerar como seria a implementação dos comandos apresentados antes. O comando create_process representa a criação de um novo processo. A gerência do processador inicializa as estruturas de dados necessárias e insere o novo processo na fila do processador. O espaço de endereçamento do novo processo deve ser igual ao espaço de endereçamento do processo que executou o comando create_process. O valor inicial para o registrador PC do novo processo deve ser o endereço fornecido como parâmetro para o comando

create_process. O valor retornado pelo create_process pode ser o identificador do novo processo.

O comando "wait_process(k)" resulta no bloqueio do processo chamador caso o processo identificado por *k* ainda exista. Nesse caso é necessário criar uma fila de processos bloqueados à espera de que o processo *k* termine. Essa fila será implementada como uma lista encadeada, associada ao próprio processo *k*. Dessa forma, quando ele terminar, parte do processo de limpeza interna do sistema operacional será liberar todos os processos que esperavam pela sua morte. Quando o processo *k* termina, é necessário armazenar essa informação. Se, depois disso, algum processo executar o comando "wait_process(k)", ele não ficará bloqueado e continuará sua execução.

O comando exit é implementado por uma chamada de sistema que resulta na destruição do processo chamador. É necessário verificar se não existe nenhum processo bloqueado por wait_process à espera da morte desse processo. Nesse caso, é necessário retirar da situação de bloqueio o processo que estava esperando e colocá-lo novamente na disputa pelo processador.

Os comandos Parbegin e Parend podem ser implementados de forma semelhante. Uma possibilidade é o sistema operacional oferecer chamadas de sistema create_process, wait_process e exit e o compilador da linguagem de programação traduzir Parbegin-Parend para uma seqüência dessas chamadas. Nesse caso, o trecho em linguagem de programação:

```
Inicio();
Parbegin
      Comando_1();
      Comando_2();
Parend;
Fim();
```

seria traduzido pelo compilador para:

```
Inicio();
f1 = create_process( Comando_1);
f2 = create_process( Comando_2);
wait_process( f1);
wait_process( f2);
Fim();
```

A especificação de paralelismo é um aspecto importante da programação concorrente, mas não é o único. Nas próximas seções serão discutidos aspectos ligados à troca de informação entre processos. Existem três formas básicas de processos trocarem informações: memória compartilhada, mensagens e arquivos. As seções 3.4, 3.5 e 3.6 tratam das soluções empregando memória compartilhada. A seção 3.7 discute o emprego de mensagens. Finalmente, as seções 3.8 e 3.9 contêm uma comparação entre as técnicas apresentadas e uma descrição do problema do *deadlock*. O emprego de arquivos para a comunicação entre processos requer o uso das técnicas de Bancos de Dados e não será abordado neste texto.

3.4 Problema da seção crítica

No exemplo do servidor de impressão, cuja organização interna aparece na Figura 3.6, cada flecha representa a passagem de dados de um processo para o outro. Como já foi dito, uma forma de implementar essa passagem de dados são variáveis compartilhadas pelos processos envolvidos na comunicação. A passagem de dados acontece quando um processo escreve em uma variável que será lida por outro processo.

A quantidade exata de memória compartilhada entre os processos pode variar conforme o programa. Processos podem compartilhar todo o seu espaço de endereçamento, apenas um segmento de memória, algumas estruturas de dados ou algumas variáveis. O sistema operacional arranja para que processos acessem as mesmas posições de memória. A Figura 3.13 ilustra duas situações: quando existe um compartilhamento total de memória (3.13a) e quando existe um compartilhamento parcial (3.13b).

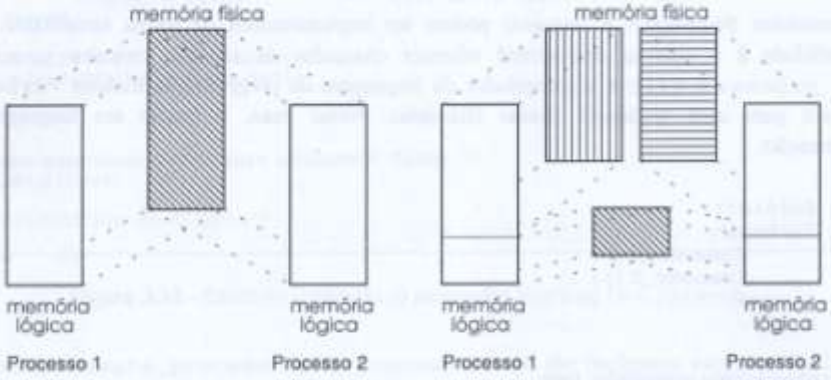

Figura 3.13 – Compartilhamento de memória.

Fazendo referência ao servidor de impressão, vamos examinar com mais detalhe a comunicação que ocorre entre o processo escritor e o processo leitor. Sempre que o processo escritor completa a escrita de um arquivo no disco, ele deve passar o nome desse arquivo para o processo leitor. É possível que, nesse momento, o processo leitor ainda não tenha concluído a leitura de arquivos recebidos anteriormente. Logo, uma forma natural de implementar essa comunicação é usar uma fila de nomes de arquivos. Ao completar a escrita de um arquivo, o processo escritor insere o nome do arquivo no fim da fila. Ao concluir a leitura de um arquivo, o processo leitor retira do início da fila o nome do próximo arquivo a ser lido. A impressão dos arquivos vai ocorrer na mesma ordem na qual eles terminaram de ser escritos no disco.

Como o tamanho da fila é variável, vamos usar uma lista encadeada para implementá-la. Cada elemento da fila será composto por dois campos: um nome de arquivo e um apontador para o próximo elemento da fila. Vamos também manter um apontador para o início da fila e outro para o fim da fila, acelerando a remoção e a inserção de elementos. Usando a linguagem C, as estruturas de dados básicas seriam:


```
struct registro{
      char nome_do_arquivo[64];
      struct registro *proximo;
      }
struct registro *inicio = NULL;        /* Aponta primeiro elemento */
struct registro *fim = NULL;           /* Aponta o último elemento */
```

As Figuras 3.14 e 3.15 mostram como seriam os códigos dos processos Escritor e Leitor. Somente as partes relativas ao manuseio da fila são mostradas. O processo Escritor testa se a fila está vazia, comparando a variável "inicio" com o valor "NULL". Caso a fila esteja vazia, tanto "inicio" como "fim" passam a apontar o novo e único elemento da fila. Caso a fila não esteja vazia, "inicio" não é alterado. O campo "proximo" do atual último elemento passa a apontar para o novo elemento, que também deve ser apontado por "fim", pois é o novo último elemento.

```
void escritor( void)
{
      struct registro *registro_novo;

      ...    /* Novo registro é apontado por "registro_novo" */

      /* Novo nome é sempre inserido no fim da fila */
      registro_novo->proximo = NULL;

      if( inicio == NULL )
            {     /* Fila vazia */
                  inicio = registro_novo;
                  fim = registro_novo;
            }
      else
            {     /* Fila não vazia */
                  fim->proximo = registro_novo;
                  fim = registro_novo;
            }
}
```

Figura 3.14 - Código do processo Escritor.

Inicialmente, o processo Leitor fica preso em um laço até que exista algum elemento na fila. Ele então retira o primeiro elemento da fila, fazendo a variável "inicio" apontar para o segundo elemento da fila, ou seja, para aquele indicado pelo campo "proximo" do atual primeiro elemento. Caso o elemento removido fosse o único da fila, seu campo "proximo" conteria "NULL", e esse valor seria copiado para "inicio". Nesse caso, o valor "NULL" também deverá ser copiado para a variável "fim".

```
void leitor( void)
{
      struct registro *ler;

      ...

      /* Espera até existir algum nome na fila */
      while( inicio == NULL )
            ;

      /* Retira o primeiro nome da fila */
      ler = inicio;
      inicio = inicio->proximo;

      /* Se era o único, acerta apontador de fim da fila */
      if( inicio == NULL )
            fim = NULL;

      ...   /* Lê o arquivo cujo nome é indicado por *ler* */
}
```

Figura 3.15 - Código do processo Leitor.

Os códigos mostrados nas Figuras 3.14 e 3.15 funcionarão perfeitamente enquanto forem executados em momentos distintos. Os problemas surgem quanto ambos os processos tentam acessar a mesma estrutura de dados ao mesmo tempo. Isso pode acontecer pois os processos Escritor e Leitor trabalham com relativa independência, e não é possível determinar antecipadamente em que instantes vão ocorrer os chaveamentos de contexto.

Vamos agora ilustrar uma seqüência de eventos capaz de gerar um erro. Suponha que a fila possui um único elemento, como mostra a Figura 3.16a. Nesse instante o processo Leitor conclui a leitura do arquivo anterior e vai retirar um novo nome de arquivo da fila. Ele inicia a executar o código mostrado na Figura 3.15. Entretanto, após executar o comando

```
if (inicio == NULL)
```

termina sua fatia de tempo e ele é suspenso. A estrutura de dados fica inconsistente (Figura 3.16b), pois o Leitor não teve tempo de concluir sua alteração.

Vamos supor que agora o processo Escritor recebe o processador. Ele concluiu a recepção de um arquivo e vai colocar o seu nome na fila. O processo Escritor executa o código mostrado na Figura 3.14 e deixa a fila como mostrado na Figura 3.16c. Observe que o novo elemento foi inserido.

Depois de algum tempo é novamente a vez do processo Leitor executar. Ele retoma sua execução do ponto em que havia sido suspenso, ou seja, do comando

```
fim = NULL;
```

Esse comando faz com que a fila adquira a forma mostrada na Figura 3.16d, o que vai resultar em um erro quando um novo elemento for inserido na fila.

Esse tipo de erro é típico de programas concorrentes. Para que o erro se manifeste, é necessária uma seqüência específica de eventos e chaveamentos de contexto. Dessa forma, o programa vai executar corretamente muitas vezes até ocorrer um erro. Caso o programador execute novamente o programa com o objetivo de identificar o erro, provavelmente ele não acontecerá. Em outras palavras, não é

fácil reproduzir um erro em programas concorrentes quando ele exige uma seqüência não trivial de eventos para ocorrer.

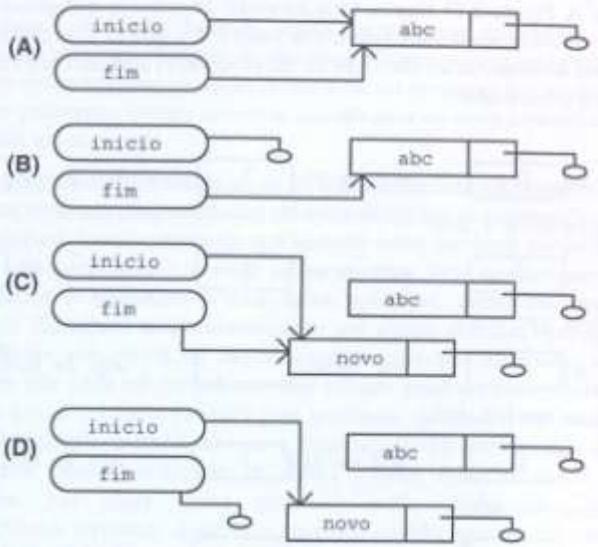

Figura 3.16 - Diversos momentos da fila de nomes.

O problema mostrado acontece porque dois processos, Escritor e Leitor, acessam a mesma estrutura de dados simultaneamente. Não é possível eliminar as variáveis compartilhadas de um programa concorrente quando elas são o mecanismo de comunicação entre os processos. A solução está em controlar o acesso dos processos a essas variáveis compartilhadas de modo a garantir que um processo não acesse uma estrutura de dados enquanto essa estiver sendo atualizada por outro processo.

Os problemas desse tipo podem acontecer de maneira muito sutil. Considere o exemplo de uma variável inteira que é incrementada por dois processos distintos. Por exemplo:

```
int X;
X = 0;
Parbegin
      ++X;
      ++X;
Parend;
```

Espera-se que o valor da variável "X" seja 2 após a execução desse trecho de programa. Entretanto, isso pode não acontecer. A variável "X" representa uma posição de memória, e a maioria dos processadores somente é capaz de incrementar um valor que está em registrador. Logo, o comando

```
++X;
```

será traduzido pelo compilador em:

```
MOVE  X, ACC      ; Move o valor de X para o acumulador
INC   ACC         ; Incrementa o conteúdo do acumulador
MOVE  ACC,X       ; Coloca o valor incrementado em X
```

que o implementa. A Figura 3.17 mostra uma sucessão de eventos que geram um erro. Após a variável X ter sido incrementada duas vezes, seu valor é 1 e não 2. É importante lembrar que o acumulador faz parte do contexto de execução de um processo e, portanto, seu valor é salvo quando um processo perde o processador.

Figura 3.17 - Problema de seção crítica envolvendo uma única variável.

Vamos chamar de **seção crítica** aquela parte do código de um processo que acessa uma estrutura de dados compartilhada. Por exemplo, o código do Escritor que insere nomes de arquivos na fila e o código do Leitor que retira esses nomes são seções críticas. O problema da seção crítica está em garantir que, quando um processo está executando sua seção crítica, nenhum outro processo entre na sua respectiva seção crítica. No exemplo, isso significa que, enquanto o processo Escritor estiver inserindo um nome na fila, o processo Leitor não poderá retirar nomes da fila, e vice-versa.

Uma solução para o problema da seção crítica estará correta quando apresentar as seguintes quatro propriedades:

- Existe exclusividade mútua entre os processos com referência a execução das respectivas seções críticas.
- Quando um processo P deseja entrar na seção crítica e nenhum outro processo está executando a sua seção crítica, o processo P não é impedido de entrar;

- Nenhum processo pode ter seu ingresso na seção crítica postergado indefinidamente, ou seja, ficar esperando para sempre;
- A solução não depende das velocidades relativas dos processos;

Soluções erradas para o problema da seção crítica normalmente apresentam a possibilidade de postergação indefinida ou a possibilidade de *deadlock*. Ocorre postergação indefinida quando um processo está preso tentando entrar na seção crítica e nunca consegue por ser sempre preterido em benefício de outros processos. Ocorre *deadlock* quando dois ou mais processos estão à espera de um evento que nunca vai acontecer.

Uma solução simples para o problema da seção crítica é desabilitar interrupções. Toda vez que um processo vai acessar variáveis compartilhadas ele antes desabilita as interrupções. Dessa forma, ele pode acessar as variáveis com a certeza de que nenhum outro processo vai ganhar o processador. No final da seção crítica, ele torna a habilitar as interrupções. Esse esquema é efetivamente usado em sistemas pequenos e dedicados a uma única aplicação, como em **sistemas embutidos** (*embedded systems*). Também é usado internamente por alguns sistemas operacionais. Entretanto, desabilitar interrupções vai contra os mecanismos de proteção discutidos no capítulo sobre multiprogramação e não pode ser considerado um método genérico para resolver o problema da seção crítica. Essa é uma operação proibida para processos de usuário em sistemas de propósito geral. Desabilitar interrupções também diminui a eficiência do sistema à medida que periféricos não são atendidos durante todas as execuções de seções críticas. Além disso, ele não funciona em máquinas paralelas, nas quais vários processos estão efetivamente sendo executados simultaneamente. Nesses sistemas, desabilitar interrupções não garante exclusividade mútua no acesso às seções críticas, pois o controle de interrupções é específico para cada processador.

Existem na literatura soluções que empregam um protocolo de acesso. Nessas soluções, os processos executam um determinado algoritmo tanto na entrada quanto na saída de sua respectiva seção crítica. Tais algoritmos não incluem nenhuma chamada ao sistema operacional, nem o emprego de instruções privilegiadas. Entretanto, eles não são muito empregados por duas razões. Primeiramente, eles são bastante complexos, o que dificulta a depuração dos programas. A segunda razão é por apresentarem a propriedade chamada *busy-waiting*. É dito que ocorre *busy-waiting* quando um processo aguarda em um laço a sinalização de um evento. O código do processo Leitor mostrado na Figura 3.15 apresenta *busy-waiting* em função do comando

```
while( inicio == NULL )
    ;
```

ou seja, o processo Leitor gasta tempo de processador sem realizar nenhum processamento útil.

Um *deadlock* pode surgir quando soluções que apresentam *busy-waiting* são empregadas em sistemas que utilizam prioridades para escalonar os processos. Voltando ao exemplo dos processos Escritor e Leitor, suponha que a fila de nomes de arquivos esteja vazia. O Leitor, ao receber o processador, tenta tirar um nome da fila mas fica preso no comando "while". Suponha agora que a prioridade do processo Leitor é superior à do processo Escritor. Nesse caso, teremos que o processo Escritor não executa, pois o processo Leitor está ocupando o processador. Por sua vez, o processo Leitor não sai do laço de espera enquanto o Escritor não inserir um nome na fila. Essa situação configura um *deadlock*, pois os dois processos vão permanecer nesse estado indefinidamente.

3.5 Spin-Lock

Uma solução possível para o problema da seção crítica é o chamado ***Spin-Lock*** ou ***Test-and-Set.*** Essa solução é baseada em uma instrução de máquina chamada "*Test-and-Set*", embora uma instrução do tipo "*Swap*" ou "*Compare on Store*" possa ser usada também. Considere uma instrução de máquina que troca (*swap*) o valor contido em uma posição de memória com o valor contido em um registrador. A posição de memória e o registrador a serem utilizados são especificados como operandos da instrução. Em resumo, temos:

Instrução SWAP(reg, mem)

[mem] → aux	"copia conteúdo da posição [mem] para um registrador auxiliar"
reg → [mem]	"copia conteúdo do registrador reg para a posição [mem] da memória"
aux → reg	"copia conteúdo do registrador auxiliar para o registrador reg"

É essencial que o processador execute toda a instrução de máquina *SWAP* sem interrupções e sem perder o acesso exclusivo ao barramento do computador. Isso é normal em máquinas com apenas um processador, mas pode exigir alguns cuidados especiais no hardware de máquinas com vários processadores acessando a mesma memória.

A seção crítica será protegida por uma variável que ocupará a posição [mem] da memória. Essa variável é normalmente chamada de *lock* (fechadura). Quando *lock* contém "0", a seção crítica está livre. Quando *lock* contém "1", ela está ocupada. A variável é inicializada com "0":

```
int l
lock = 0;  /* Define variável e inicializa com 0,
                        seção crítica livre */
```

Antes de entrar na seção crítica, um processo precisa "fechar a porta", colocando "1" em *lock*. Entretanto, ele só pode fazer isso se "a porta estiver aberta". Logo, antes de entrar na seção crítica, o processo executa o seguinte código:

```
do {        reg = 1;                /* Entrada da seção crítica */
            swap( reg, lock);
            } while( reg == 1);
código-da-seção-crítica
```

Observe que o processo fica repetidamente colocando "1" em *lock* e lendo o valor que estava lá antes. Se esse valor era "1", a seção estava (e está) ocupada, e o processo repete a operação. Quando o valor lido de *lock* for "0", então a seção crítica está livre, o processo sai do `do-while` e pode prosseguir sua execução dentro da seção crítica.

Ao sair da seção crítica, basta colocar "0" na variável *lock*. Se houver algum processo esperando para entrar, a sua instrução *swap* pegará o valor "0", e ele entrará.

```
lock = 0;           /* Saída da seção crítica */
```

A vantagem do *Spin-Lock* é sua simplicidade, aliada ao fato de que não é necessário desabilitar interrupções. A instrução de máquina necessária está presente em praticamente todos os processadores atuais. Essa mesma solução funciona em máquinas com vários processadores, a partir de alguns cuidados na construção do hardware.

Entretanto, *Spin-Lock* tem como desvantagem o *busy-waiting*. O processo que está no laço de espera executando "*swap*" ocupa o processador enquanto espera. Além disso, existe a possibilidade de postergação indefinida, quando vários processos estão esperando simultaneamente para ingressar

na seção crítica e um processo "muito azarado" sempre perde na disputa de quem "pega antes" o valor "0" colocado na variável *lock*.

Na prática o *spin-lock* é muito usado em situações nas quais a seção crítica é pequena (algumas poucas instruções). Nesse caso, a probabilidade de um processo encontrar a seção crítica ocupada é baixíssima, o que torna o *busy-waiting* e a postergação indefinida situações teoricamente possíveis, mas altamente improváveis.

3.6 Semáforos

Um mecanismo de sincronização entre processos muito empregado é o **semáforo**. Ele foi criado pelo matemático holandês E. W. Dijkstra em 1965. O semáforo é um tipo abstrato de dado composto por um valor inteiro e uma fila de processos. Somente duas operações são permitidas sobre o semáforo. Elas são conhecidas como **P** (do holandês *proberen*, testar) e **V** (do holandês *verhogen*, incrementar).

Quando um processo executa a operação P sobre um semáforo, o seu valor inteiro é decrementado. Caso o novo valor do semáforo seja negativo, o processo é bloqueado e inserido no fim da fila desse semáforo. Quando um processo executa a operação V sobre um semáforo, o seu valor inteiro é incrementado. Caso exista algum processo bloqueado na fila desse semáforo, o primeiro processo da fila é liberado. Podemos sintetizar o funcionamento das operações P e V sobre o semáforo S da seguinte forma:

```
P(S):
     S.valor = S.valor - 1;
     Se  S.valor < 0
          Então bloqueia o processo, insere em S.fila
V(S):
     S.valor = S.valor + 1;
     Se S.fila não está vazia
          Então retira processo P de S.fila, acorda P
```

Para que semáforos funcionem corretamente, é essencial que as operações P e V sejam atômicas. Isso é, uma operação P ou V não pode ser interrompida no meio e outra operação sobre o mesmo semáforo iniciada.

Semáforos tornam a proteção da seção crítica muito simples. Para cada estrutura de dados compartilhada, deve ser criado um semáforo S inicializado com o valor 1. Todo processo, antes de acessar essa estrutura, deve executar um P(S), ou seja, a operação P sobre o semáforo S associado com a estrutura de dados em questão. Ao sair da seção crítica, o processo executa V(S).

Inicialmente a seção crítica está livre, e o valor de S é 1. O primeiro processo a chegar executa P(S). O novo valor de S é zero, e o processo continua sua execução, entrando assim na seção crítica. O segundo processo a chegar também executa P(S). Mas agora o valor de S ficou –1, e o segundo processo é bloqueado. Caso um terceiro processo tente acessar a estrutura de dados, ele também executa P(S), S passa a valer –2, e o terceiro processo também é bloqueado. É interessante observar que, se o valor de um semáforo é negativo, isso significa que existem processos na fila de espera desse semáforo e, mais, que o valor absoluto do semáforo é igual ao número de processos na fila de espera.

Quando o primeiro processo sai da seção crítica, ele executa V(S). Isso faz com que o valor de S passe para -1 e o segundo processo seja liberado. O segundo processo acessa a estrutura de dados e

também executa V(S), fazendo com que o valor de S passe a ser zero e liberando o terceiro processo. Quando finalmente o terceiro processo sai da seção crítica, ele executa V(S), e S volta a valer 1. Nesse caso nenhum processo é liberado, pois a fila do semáforo está vazia, e voltamos à situação original.

Por exemplo, o problema do incremento simultâneo de uma variável poderia ser resolvido por:

```
int X = 0;
semaphore S = 1;

Parbegin
     Begin P(S);
           ++X;
           V(S);
     End
     Begin P(S);
           ++X;
           V(S);
     End
Parend;
```

Além de resolver o problema da exclusão mútua, semáforos também podem ser usados para estabelecer relações de precedência entre processos. Por exemplo, imagine dois processos P0 e P1, sendo que P0 executa as funções `p0rot1()` e `p0rot2()`, enquanto P1 executa as rotinas `p1rot1()` e `p1rot2()`. Suponha que, em função da aplicação, `p1rot2()` somente possa ser executada depois de `p0rot1()`. Podemos resolver essa situação, usando um semáforo para bloquear P1 até que P0 tenha terminado a função `p0rot1()`. A solução ficaria:

```
semaphore S = 0;
Parbegin
     Begin
          p0rot1();          /* processo P0 */
          V(S);
          p0rot2();
     End
     Begin
          p1rot1();          /* processo P1 */
          P(S);
          p1rot2();
     End
Parend;
```

Existem algumas possíveis variações no que diz respeito às operações P e V. Algumas implementações de semáforos não permitem que ele assuma valores negativos. Nesse caso, o processo que executa um P fica bloqueado quando o valor do semáforo já for zero. Também é possível ordenar a fila do semáforo segundo algum esquema de prioridade ao invés de pela ordem de chegada. Finalmente, algumas implementações oferecem, além de P e V, uma operação de consulta que retorna o valor atual do semáforo sem alterá-lo.

Uma variação muito comum de semáforos são as construções ***mutex*** ou **semáforo binário.** Nesse caso, temos um semáforo capaz de assumir apenas os valores 0 e 1. Ele pode ser visto como uma variável tipo mutex, a qual assume apenas os valores **livre** e **ocupado**. Nesse caso, as operações P e V são normalmente chamadas de ***lock*** e ***unlock***, respectivamente. Assim como P e V, elas devem ser atômicas. Sua operação é simples:


```
lock(x):
     Se x está livre
          Então     marca x como ocupado
          Senão     insere processo no fim da fila "x"
unlock(x):
     Se fila "x" está vazia
          Então     marca x como livre
          Senão     libera processo do início da fila "x"
```

O problema da seção crítica pode ser facilmente resolvido com o mutex:

```
mutex x = LIVRE;
...
lock(x);          /* entrada da seção crítica */
Seção Crítica;
unlock(x);        /* saída da seção crítica */
```

3.6.1 Implementação de semáforos

Vamos agora examinar a implementação das operações sobre semáforos. Cada semáforo pode ser implementado como um registro composto por um valor inteiro e um apontador para descritor de processo. Quando um processo executa a operação P(S), o registro correspondente ao semáforo S é localizado. O valor inteiro é decrementado. Caso o novo valor do semáforo seja negativo, o descritor do processo em questão é removido da fila do processador e inserido na fila do semáforo S. Observe que esse processo é o primeiro da fila do processador, ou seja, o processo que estava em execução. Caso o valor do semáforo não tenha ficado negativo, o processo continua sua execução normalmente.

Quando um processo executa a operação V(S), o registro correspondente ao semáforo S é localizado. O valor inteiro é incrementado. Caso exista algum descritor de processo na fila do semáforo S, o primeiro descritor de processo é removido da fila do semáforo e inserido na fila do processador. Isso faz com que o processo volte a disputar processador, ou seja, o processo é desbloqueado.

A atomicidade das operações P e V é obtida facilmente através da desabilitação das interrupções enquanto o registro referente ao semáforo é manipulado. Como essa implementação de P e V requer acesso aos descritores de processo e também a capacidade de desabilitar interrupções, elas devem ser implementadas como chamadas de sistema. Chamadas de sistema adicionais podem ser fornecidas para que processos possam criar e destruir semáforos.

Em sistemas com vários processadores, não basta desabilitar interrupções para garantir a atomicidade das operações P e V. Enquanto o processo no processador A estiver acessando o semáforo S, um outro processo no processador B poderá fazer o mesmo, comprometendo a correta execução da operação. Nesse caso, a solução usual é considerar as próprias operações P e V como seções críticas dentro do sistema operacional e empregar uma solução tipo *spin-lock* para sua execução. As seções críticas dos programas de usuário continuariam a ser protegidas com o uso de semáforo. Embora soluções tipo *spin-lock* apresentem *busy-waiting*, as operações P e V são rápidas, e os processos não ficam esperando muito tempo. As seções críticas dos programas de usuário, potencialmente demoradas, seriam protegidas por semáforos, os quais não apresentam *busy-waiting*.

3.6.2 Problema do produtor-consumidor

É comum, em programas concorrentes, a existência de um processo que produz continuamente informações que serão usadas por outro processo. Essa situação é chamada de **problema do produtor-consumidor**. O dado produzido e consumido pode ser um número, um string, um registro, etc. Entre o produtor e o consumidor existe um *buffer* no qual os dados já produzidos mas ainda não consumidos são armazenados temporariamente. Esse *buffer* é tipicamente implementado de maneira circular, como ilustrado pela Figura 3.18. Apontadores mostram o próximo local de inserção e de remoção de dados do *buffer*. Dessa forma, produtor e consumidor podem trabalhar em paralelo. Quando o produtor tenta colocar um dado no *buffer* e esse está cheio, o produtor deve ser bloqueado até que o consumidor retire algo do *buffer*, liberando uma entrada. O mesmo acontece com o consumidor quando esse tenta retirar um dado do *buffer* vazio.

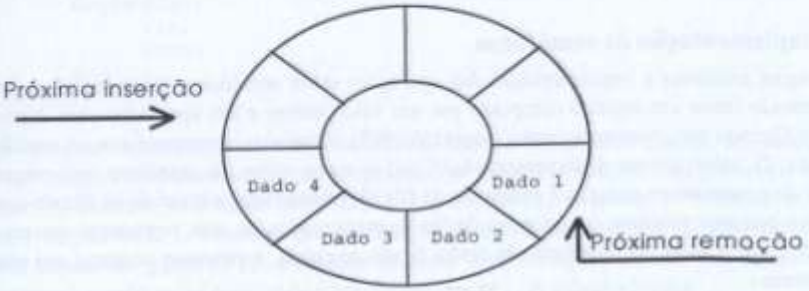

Figura 3.18 - *Buffer* circular com 4 entradas ocupadas.

A Figura 3.19 mostra uma implementação desse problema utilizando semáforos. Um *buffer* com capacidade para N dados é implementado como um vetor de N elementos. O tipo exato do dado em questão é irrelevante. As variáveis "proxima_insercao" e "proxima_remocao" indicam onde, no vetor, deve ser feita a próxima inserção e a próxima remoção de dados, respectivamente. O efeito circular do *buffer* é obtido através da forma como as variáveis "proxima_insercao" e "proxima_remocao" são incrementadas. Após valer N-1, elas voltam a apontar a entrada zero do vetor. O operador "%" representa a operação "resto da divisão".

A solução mostrada na Figura 3.19 utiliza 3 semáforos. O semáforo "exclusao_mutua" é usado para garantir que somente um processo acessa as variáveis compartilhadas de cada vez. O semáforo "espera_dado" é usado para bloquear o processo consumidor quando esse encontra o *buffer* vazio. O semáforo "espera_vaga" é usado para bloquear o processo produtor quando ele encontra o *buffer* cheio. Observe que "exclusao_mutua" é inicializado com 1, para permitir o ingresso de apenas um processo na seção crítica. O semáforo "espera_dado" é inicializado com zero, pois inicialmente não existe nenhum dado no *buffer*, e o consumidor deve ser bloqueado caso tente retirar algo do *buffer*. Esse semáforo é incrementado sempre que o produtor acrescenta um dado ao *buffer*. O semáforo "espera_vaga" é inicializado com N, pois inicialmente existem N entradas vagas no *buffer*. O processo produtor somente deverá ser bloqueado após essas N entradas terem sido ocupadas. Isso se, durante esse intervalo de tempo, o processo consumidor não liberar nenhuma entrada e incrementar o semáforo. A solução mostrada na Figura 3.19 é igualmente correta para o caso em que existem vários processos produtores e vários processos consumidores.


```
struct tipo_dado buffer[N];
int proxima_insercao = 0;
int proxima_remocao = 0;
...
semaphore exclusao_mutua = 1;
semaphore espera_vaga = N;
semaphore espera_dado = 0;
...
void produtor( void)
{       ...
        P( espera_vaga );
        P( exclusao_mutua );
        buffer[ proxima_insercao ] = dado_produzido;
        proxima_insercao = ( proxima_insercao + 1 ) % N;
        V( exclusao_mutua );
        V( espera_dado );
        ...
}
...
void consumidor( void)
{       ...
        P( espera_dado );
        P( exclusao_mutua );
        dado_a_consumir = buffer[ proxima_remocao ];
        proxima_remocao = ( proxima_remocao + 1 ) % N;
        V( exclusao_mutua );
        V( espera_vaga );
        ...
}
```

Figura 3.19 - Problema do Produtor-Consumidor com semáforos.

3.7 Mensagens

As seções anteriores mostraram como um programa concorrente pode ser construído a partir de variáveis compartilhadas e um mecanismo de sincronização apropriado. Nesta seção, vamos considerar programas concorrentes construídos a partir da troca de mensagens entre processos. Uma aplicação imediata para esse mecanismo está nos sistemas distribuídos. Nesse caso, processos executando em diferentes máquinas não possuem variáveis compartilhadas. Eles trocam informações através de mensagens via uma rede de comunicação.

Tipicamente, o mecanismo que permite a troca de mensagens entre processos é implementado pelo sistema operacional. Ele é acessado através de duas chamadas de sistema básicas: `Send` e `Receive`.

A chamada de sistema **Send** possui pelo menos dois parâmetros: a mensagem a ser enviada e o destinatário da mensagem. Existem dois tipos de endereçamento. No endereçamento direto, a mensagem é endereçada explicitamente a um processo em particular. O processo remetente deve conhecer o identificador do processo destinatário. Dessa forma, o processo remetente somente pode enviar mensagens para os processos dos quais ele conhece o identificador ou processos que possuem um identificador público. Em geral, isso não é um problema pois os processos envolvidos na comunicação fazem parte do mesmo programa.

Alguns sistemas trabalham com endereçamento indireto. Nesse caso, o sistema operacional implementa um recurso chamado de **caixa postal**. Mensagens não são enviadas para processos, mas sim para caixas postais. As mensagens deverão ser retiradas da caixa postal por outro processo. O sistema operacional normalmente fornece chamadas de sistema que permitem a criação e a

destruição de caixas postais. Também é o sistema operacional que controla quais processos possuem o direito de ler ou escrever na caixa postal. O processo remetente somente pode enviar mensagens para as caixas postais das quais ele conhece o identificador, ou para as caixas postais que possuem um identificador público. Além disso, ele deve possuir o direito de escrita sobre a caixa postal em questão.

A chamada de sistema **`Receive`** possui pelo menos um parâmetro: o endereço da variável do processo onde deverá ser colocada a mensagem lida. Em alguns sistemas, todas as mensagens recebidas pelo processo são enfileiradas, e a chamada `Receive` retorna a primeira mensagem da fila. Opcionalmente, a chamada `Receive` possui como segundo parâmetro o identificador do processo do qual se deseja receber a mensagem. Nesse caso, a chamada `Receive` somente será satisfeita com uma mensagem do processo especificado como parâmetro.

Quando endereçamento indireto é usado, a chamada `Receive` possui como segundo parâmetro o identificador da caixa postal em questão. Um processo somente pode retirar mensagens das caixas postais das quais ele conhece o identificador, ou de caixas postais que possuem um identificador público. Além disso, ele deve possuir o direito de leitura sobre a caixa postal em questão.

Diversas variações são possíveis a partir dos esquemas descritos acima. Por exemplo, mensagens podem ter prioridades. Nesse caso, elas são enfileiradas conforme a sua respectiva prioridade, e o `Receive` sempre retorna a mensagem de mais alta prioridade que está disponível. Existem ainda três comportamentos possíveis para o `Receive` quando não existe uma mensagem disponível para o processo. O processo pode ficar bloqueado até que chegue uma mensagem que satisfaça o `Receive`. O processo pode retornar imediatamente com uma indicação de falha. Ou ainda, o processo pode ficar bloqueado por algum tempo esperando uma mensagem e retornar ao final desse com uma indicação de falha, caso nenhuma mensagem tenha chegado.

É importante observar que caixas postais permitem uma comunicação de N para N processos, enquanto o endereçamento direto permite uma comunicação de 1 para 1 processo. Existem sistemas que permitem a criação de **grupos de processos**. Nesse caso, um processo pode enviar uma mensagem para um grupo de processos, ao invés de para um processo em particular. Existem várias semânticas possíveis para esse envio: todos os processos do grupo recebem uma cópia da mensagem; apenas um processo qualquer do grupo recebe a mensagem; pelo menos um processo do grupo recebe a mensagem. É possível que o sistema operacional suporte mais de uma semântica, o que exige que o processo remetente informe a semântica desejada como um terceiro parâmetro do `Send`.

Suponha agora que endereçamento direto é usado, e o processo P enviou uma mensagem para o processo Q, mas o processo Q ainda não executou o `Receive`. Alguns sistemas oferecem *buffers* para as mensagens que foram enviadas mas ainda não foram recebidas. Nesse caso, o processo remetente retorna imediatamente da chamada `Send`, e o sistema operacional armazena a mensagem. Ela será entregue quando o processo destinatário executar um `Receive`. Como os recursos do sistema são limitados, uma solução alternativa é armazenar até um determinado limite, medido em número de mensagens ou em bytes. Quando esse limite é atingido, o sistema operacional passa a bloquear os processos remetentes cujos respectivos destinatários ainda não executaram o `Receive`.

No outro extremo temos os sistemas que não oferecem *buffers* para as mensagens. Nesse caso, o remetente é sempre bloqueado até o respectivo destinatário executar um `Receive`. Alguns autores chamam esse tipo de comunicação de ***Rendezvous***, porque a comunicação só ocorre quando ambos

os processos "se encontram". Nesse esquema, o primeiro processo que chega no ponto de comunicação espera pelo outro.

A discussão sobre buferização é válida tanto para sistemas que empregam endereçamento direto quanto para aqueles baseados em endereçamento indireto. Os mecanismos de mensagens com buferização são mais flexíveis e tendem a resultar em programas concorrentes mais eficientes. Entretanto, a necessidade de prover buferização para as mensagens torna o sistema operacional mais complexo, além de consumir recursos, na forma de tabelas e áreas de memória para manter as mensagens.

Uma forma popular de usar mensagens é a **chamada remota de procedimento** (**RPC**, *remote procedure call*). Nesse caso, o processo chamador utiliza o `Send` para solicitar a execução de uma determinada rotina em outro processo, passando os parâmetros para essa rotina como parte da mensagem. Em seguida, ele executa um `Receive`, para receber os resultados da execução da rotina. No outro lado, um processo executa um `Receive` para receber o pedido de execução de uma rotina, executa a rotina com os parâmetros recebidos na mensagem e utiliza o `Send` para enviar os resultados da execução ao processo chamador. Dessa forma, a comunicação entre processos baseada em mensagens é disfarçada de uma simples chamada de sub-rotina.

Em sistemas distribuídos, podem ocorrer erros na rede de comunicação. É responsabilidade dos protocolos de comunicação prover mecanismos para a recuperação de faltas simples como, por exemplo, a perda de uma mensagem. Em um caso extremo, a máquina na qual está um dos processos envolvidos na comunicação pode parar completamente. Nesse caso, o sistema operacional deve ter o cuidado de detectar a situação e não deixar outros processos envolvidos na comunicação bloqueados para sempre.

Vamos agora voltar ao exemplo do produtor-consumidor apresentado antes. Uma implementação simples é fazer com que o produtor envie mensagens com os dados para o consumidor. Supondo endereçamento direto, o produtor executa o comando

```
Send ( id_processo_consumidor, dado_a_enviar )
```

enquanto o consumidor executa o comando

```
Receive( id_processo_produtor, dado_recebido )
```

e a própria buferização do sistema operacional é utilizada.

Quando um controle explícito da buferização é necessário, podemos usar um processo intermediário entre o produtor e o consumidor. O produtor envia um dado para o processo intermediário e espera por uma confirmação de que o dado foi armazenado. O código do produtor fica:

```
Send ( id_processo_intermediario, dado_a_enviar )
Receive( id_processo_intermediario, confirmacao )
```

Por sua vez, o consumidor sempre envia uma solicitação de dados ao processo intermediário, que envia o dado assim que houver um disponível. O código do consumidor fica:

```
Send ( id_processo_intermediario, pedido )
Receive( id_processo_intermediario, dado_recebido )
```

O processo intermediário executa um laço no qual fica esperando dados do produtor ou pedidos do consumidor. Ele mantém localmente um *buffer* circular para armazenar as mensagens que o produtor já enviou mas o consumidor ainda não pediu. Não existe problema de seção crítica com esse *buffer*, pois o processo intermediário é o único com direito de acessá-lo. O código do processo

intermediário é mostrado na Figura 3.20. Observe que a operação `receive()` nesse caso aceita mensagens de qualquer processo, colocando a identificação do processo remetente na variável `id_processo_remetente`.

```
produtor_esperando = false
consumidor_esperando = false

while( true )
   {  Receive( &id_processo_remetente, mensagem )

      if( id_processo_remetente == id_processo_produtor )   /* do produtor */
            if( consumidor_esperando )
              {   consumidor_esperando = false
                  Send( id_processo_consumidor, mensagem )
                  Send( id_processo_produtor, confirmacao )
                  }
            else{ coloca mensagem no buffer circular
                  if( buffer cheio )
                        produtor_esperando = true
                  else  Send( id_processo_produtor, confirmacao)
                  }

      else  if( produtor_esperando )                        /* do consumidor */
              {   Retira dado do buffer circular
                  Send( id_processo_consumidor, dado_do_buffer )
                  produtor_esperando = false
                  Send( id_processo_produtor, confirmacao )
                  }
            else{ if( buffer vazio )
                        consumidor_esperando = true
                  else{ Retira dado do buffer circular
                        Send( id_processo_consumidor, dado_do_buffer)
                        }
      }
```

Figura 3.20 - Processo intermediário no problema dos produtores e consumidores.

Nas duas situações consideradas, foi suposta a existência de um único produtor e de um único consumidor. Vamos supor agora que existam N produtores e N consumidores. Quando um produtor deseja enviar um dado ele deve determinar qual dos consumidores está livre no momento. Isso pode ser feito através de um processo intermediário, semelhante ao usado antes. Agora o processo intermediário seria responsável por manter uma tabela com a informação "quais consumidores estão livres" e então direcionar o dado que chega para um deles. O problema de identificar um consumidor livre é automaticamente resolvido quando caixas postais são usadas para a comunicação. As mensagens são enviadas para a caixa postal e ficam enfileiradas. Quando um consumidor fica livre, ele lê uma das mensagem da caixa postal.

3.8 Visão geral e comparação

Esta seção contém comentários sobre as técnicas discutidas anteriormente para a comunicação entre processos. Como visto no início desse capítulo, um programa concorrente é formado por processos que necessitam trocar informações. Essa troca de informações pode ocorrer através de variáveis compartilhadas ou de mensagens.

Nas soluções baseadas em variáveis compartilhadas, o sistema operacional oferece algum mecanismo de sincronização auxiliar, como semáforos. Também é necessário que a gerência de memória permita o acesso de dois processos a uma mesma memória física. Quando mensagens são empregadas, o sistema operacional deve implementar algum mecanismo de troca de mensagens para ser usado pelos processos.

Essas duas soluções (memória compartilhada e mensagens) são equivalentes no sentido de que qualquer programa concorrente pode ser implementado com uma ou com outra forma. Em geral, memória compartilhada é mais eficiente que troca de mensagens. Com memória compartilhada não existe a necessidade de copiar os dados da memória do remetente para uma área do sistema operacional e depois para a memória do destinatário. Os processos alteram diretamente as variáveis compartilhadas. Entretanto, em sistemas distribuídos, mensagens são a forma natural de comunicação.

Muitos sistemas operacionais oferecem os dois mecanismos. Nesses sistemas, processos na mesma máquina podem compartilhar memória. A forma como a memória compartilhada é implementada depende da gerência de memória empregada, sendo relativamente simples em sistemas que contam com paginação ou segmentação. A implementação local de semáforos também já foi discutida. Processos também podem usar mensagens para comunicação local ou remota. A implementação de um sistema de mensagens sem buferização pode ser feita através da manipulação dos descritores de processos. Um sistema de mensagens com buferização implica a reserva de memória para armazenar mensagens e algoritmos para gerenciar essa memória. Além disso, é necessário liberar automaticamente áreas de memória ocupadas por mensagens cujos processos destinatários já foram destruídos. Finalmente, a implementação de mensagens entre máquinas diferentes exige a existência de um subsistema de comunicação que implemente os protocolos de comunicação necessários.

Nesse capítulo, foram discutidos apenas os conceitos básicos da programação concorrente. Não foram abordados vários mecanismos importantes, como os **monitores**, nos quais o conceito de tipo abstrato de dado é usado para encapsular variáveis compartilhadas e tornar a programação mais fácil. Também existem na literatura diversos problemas clássicos de programação concorrente, tais como os "**filósofos jantadores**" e os "**leitores e escritores**". Existem livros que tratam especificamente desse assunto, como [TOS01]. A programação concorrente usando a linguagem de programação Java é discutida em [LEA97]. Uma descrição da biblioteca ***Pthreads***, usada para programação concorrente em ambiente **UNIX POSIX** pode ser encontrada em [NIC96]. Finalmente, existe o livro clássico [AND91], que discute diversos esquemas para programação de aplicações concorrentes.

3.9 Deadlock

Embora ***deadlocks*** possam ocorrer em diversos pontos de um sistema operacional, eles são um dos principais problemas dos programas concorrentes e, por isso, serão discutidos nesta seção. Também é comum a ocorrência de *deadlocks* envolvendo arquivos e periféricos.

Por definição, um conjunto de N processos está em *deadlock* quando cada um dos N processos está bloqueado à espera de um evento que somente pode ser causado por um dos N processos do conjunto. Obviamente, essa situação somente pode ser alterada por alguma iniciativa que parta de um processo fora do conjunto dos N processos.

A Figura 3.21 ilustra graficamente uma situação de *deadlock*. Círculos representam processos, e quadrados representam recursos. Uma flecha do processo para o recurso significa que o processo

está bloqueado à espera daquele recurso. Uma flecha do recurso para o processo significa que aquele recurso foi alocado ao processo.

No exemplo da Figura 3.21, o processo P5 está bloqueado à espera do recurso R3. Entretanto, ele não está em *deadlock*, pois o processo P2 não está bloqueado. Ele pode executar até o fim, liberar o recurso R3, e então o processo P5 poderá executar. O mesmo não ocorre com os processos P1, P3 e P4. O processo P4 está bloqueado à espera do recurso R2, ocupado pelo processo P1. Os processos P1 e P3 estão bloqueados à espera do recurso R1, ocupado pelo processo P4. Logo, P1, P3 e P4 estão em *deadlock*.

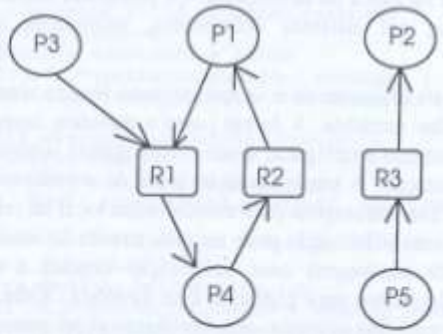

Figura 3.21 - Gráfico representando processos e recursos.

Existem quatro condições necessárias para ser possível a ocorrência de um *deadlock* em determinado sistema. São elas:

- Existência de recursos que precisam ser acessados de forma exclusiva;
- Possibilidade de processos manterem recursos alocados enquanto esperam por recursos adicionais;
- Necessidade de os recursos serem liberados pelos próprios processos que os estão utilizando;
- Possibilidade da formação de uma espera circular do tipo: o processo P_1 espera pelo recurso R_1 que está com o processo P_2, que espera pelo recurso R_2 que está com o processo P_3, etc, e P_N espera pelo recurso R_N que está com o processo P_1.

Existem várias formas de tratar o problema do *deadlock*. É impossível a ocorrência de um *deadlock* no sistema se, para cada recurso, for eliminada pelo menos uma das quatro condições necessárias. Não é necessário eliminar a mesma condição para todos os recursos, mas eliminar para cada recurso uma das condições listadas. Infelizmente, isso não é facilmente realizável. Por exemplo, no caso do recurso "processador" a terceira condição não é verdadeira, pois o processo perde o processador ao ficar bloqueado por uma razão qualquer, desde que não exista *busy-waiting*. Outro exemplo clássico é um programa concorrente onde existem diversos semáforos. Pode-se numerar os semáforos e exigir que os processos somente executem a operação "P" sobre semáforos cujo

número é maior do que o número de quaisquer semáforos que tenham sido obtidos antes. Desta forma, a quarta condição torna-se impossível quando os recursos são apenas os semáforos.

Outra forma de tratar o problema do *deadlock* é evitar a ocorrência através do cuidado de, para cada pedido de alocação de recursos, o sistema analisar as implicações de atendê-lo. Essa técnica implica subutilização dos recursos do sistema e em *overhead* de execução. A teoria sobre *deadlocks* é extensa e foge do escopo deste livro. Maiores informações podem ser obtidas em [TOS03].

Uma forma usual é deixar acontecer o *deadlock*, detectá-lo e então eliminá-lo. A detecção pode ser feita de forma automática ou manual. O *deadlock* é eliminado através da destruição dos processos envolvidos e da liberação dos respectivos recursos. A quase totalidade dos sistemas operacionais utiliza essa abordagem, o que na prática significa que nenhuma técnica especial com respeito aos *deadlocks* é utilizada. O mesmo não acontece nos sistemas gerenciadores de bancos de dados, nos quais cuidados especiais são tomados a esse respeito.

3.10 Exercícios

1) Desenhe o grafo de precedência relativo ao código das Figuras 3.11 e 3.12. (Seção 3.3)

2) Desenhe o grafo de precedência do código abaixo (Seção 3.3):

```
main()
{
[1]   int f1, f2, f3;   /* Identifica processos filho*/
[2]   printf("Alo do pai\n");
[3]   f1 = create_process( codigo_do_filho );          /* Cria filho 1 */
[4]   printf("Filho 1 criado\n");
[5]   f2 = create_process( codigo_do_filho );          /* Cria filho 2 */
[6]   printf("Filho 2 criado\n");
[7]   wait_process( f1);
[8]   printf("Filho 1 morreu\n");
[9]   f3 = create_process( codigo_do_filho );          /* Cria filho 3 */
[10]  printf("Filho 3 criado\n");
[11]  wait_process( f3);
[12]  printf("Filho 3 morreu\n");
[13]  wait_process( f2);
[14]  printf("Filho 2 morreu\n");
[15]  exit();
}
codigo_do_filho()
{
[16]  printf("Alo do filho\n");
[17]  exit();
}
```

3) Descreva o erro na implementação do produtor-consumidor mostrada abaixo. Crie uma seqüência de eventos que termina em algum comportamento indesejado para o programa. (Seção 3.6)

```
struct tipo_dado buffer[N];
int proxima_insercao = 0;
int proxima_remocao = 0;
...
semaphore exclusao_mutua = 1;
semaphore espera_vaga = N;
semaphore espera_dado = 0;
...
void produtor( void)
{     ...
      P( exclusao_mutua );
      P( espera_vaga );
      buffer[ proxima_insercao ] = dado_produzido;
      proxima_insercao = ( proxima_insercao + 1 ) % N;
      V( exclusao_mutua );
      V( espera_dado );
      ...
}
...
void consumidor( void)
{     ...
      P( exclusao_mutua );
      P( espera_dado );
      dado_a_consumir = buffer[ proxima_remocao ];
      proxima_remocao = ( proxima_remocao + 1 ) % N;
      V( exclusao_mutua );
      V( espera_vaga );
      ...
}
```

4) Pesquise a literatura a respeito dos mecanismos "mutex" e "variáveis condição" do POSIX. Implemente as operações P e V em um semáforo usando aquelas duas construções básicas do POSIX. As operações P e V deverão ser substituídas por um código C com semântica similar. Lembre-se de que essas operações devem ser atômicas.

5) Mostre, através de exemplos, que a solução apresentada na Figura 3.19 para o problema do produtor-consumidor suporta múltiplos processos produtores e múltiplos processos consumidores. (Seção 3.6)

6) Simplifique a solução apresentada na Figura 3.19 para o problema do produtor-consumidor para o caso de existir apenas uma vaga disponível no buffer circular. (Seção 3.6)

7) Localize, na literatura sobre programação concorrente, a descrição do "problema dos filósofos jantadores" e implemente uma solução usando semáforos. Faça o mesmo com o "problema dos leitores e escritores".

8) O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

É possível vários leitores lerem ao mesmo tempo ?

É possível vários escritores escreverem ao mesmo tempo ?

É possível postergação indefinida de um escritor ?

É possível leitores e escritores acessarem ao mesmo tempo ?

```
int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;
```

```
void leitor( void)
{     ...
       P( exclusivo );
       if( nl > 0 )     ++ nl;
       else{
              P( tipo );
              nl = 1;
              }
        V( exclusivo );
        Acessa o texto
        P( exclusivo );
         -- nl;
        if( nl == 0 )  V( tipo );
        V( exclusivo );
         ...
}
```

```
void escritor( void)
{     ...
         P( exclusivo );
         Acessa o texto
         V( exclusivo );
      ...
}
```

9) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor de Semáforo. Suponha que o processo S será o servidor, implementando a funcionalidade de um único semáforo inicializado com 1. Para executar a operação P, um processo Cliente envia uma mensagem "op_p" para o Servidor que responde "ok" apenas quando o Cliente estiver liberado para prosseguir. No caso de uma operação V, apenas o Cliente envia a mensagem "op_v" para o Servidor, mas não existe mensagem de resposta. Mostre o algoritmo do processo S, em português estruturado. Supor "receive" bloqueante e a existência de vários clientes. (Seção 3.7)

10) Imagine formas de tornar impossível alguma das quatro condições necessárias para a ocorrência de um *deadlock* em determinado sistema. Essa forma deve ser tal que, uma vez aplicada, *deadlocks* não acontecerão. Discuta a viabilidade prática de sua solução. (Seção 3.9)

11) O problema dos leitores/escritores consiste de um texto que pode ser lido ou escrito por vários processos. Considerando o código abaixo, responda justificando:

É possível vários leitores lerem ao mesmo tempo ?

É possível vários escritores escreverem ao mesmo tempo ?

É possível postergação indefinida de um leitor ?

É possível postergação indefinida de um escritor ?

```
int nl = 0;
semaphore tipo = 1;
semaphore exclusivo = 1;
```

```
void leitor( void)
{    ...
       P( exclusivo );
       if( nl > 0 )      ++ nl;
       else{
             P( tipo );
             nl = 1;
             }
        V( exclusivo );
        Acessa o texto
        P( exclusivo );
         -- nl;
        if( nl == 0 )  V( tipo );
        V( exclusivo );
         ...
}
```

```
void escritor( void)
{    ...
       P( tipo );
       Acessa o texto
       V( tipo );
     ...
}
```

12) Em um sistema que suporta programação concorrente apenas através da troca de mensagens, será criado um Servidor para controlar o uso das portas seriais. Quando um processo Cliente deseja usar uma porta serial, ele envia uma mensagem "Aloca" para o Servidor. Existem N portas seriais, todas equivalentes, mas cada uma pode ser usada somente por um Cliente de cada vez. O Servidor informa ao Cliente a porta que ele vai usar através da mensagem "Porta p". Ao concluir o uso, o Cliente envia para o Servidor a mensagem "Libera p". Suponha que exista mais do que N processos Clientes. Mostre o algoritmo do Servidor, em português estruturado. Supor "receive" bloqueante. (Seção 3.7)

4

Gerência do Processador

Na descrição de multiprogramação, o conceito de processo foi apresentado de forma abstrata. Foram abordados os conceitos de ciclos e estados de processo e o relacionamento entre os processos. Também foi discutida a necessidade de filas para administrar a alocação de recursos aos processos. Neste capítulo, será discutida a implementação do conceito de processo. Como os processos são uma conseqüência direta do compartilhamento do processador, é natural estudar a implementação de processos juntamente com os algoritmos para escalonamento do processador.

4.1 Bloco descritor de processo

Como foi visto, na multiprogramação, processos são interrompidos e mais tarde continuados. Por exemplo, suponha que o processo 1, executando, faça uma chamada de sistema. Ele é bloqueado, e um processo 2 passa a executar. Quando ocorrer a interrupção causada pelo periférico, uma possibilidade é suspender o processo 2 e retomar o processo 1. A maioria dos processos que estão na fila de aptos já executaram algum tempo. Eles esperam para receber o processador novamente. A única exceção são os processos novos, que entram no primeiro ciclo de processador.

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. Abaixo está uma lista de campos normalmente encontrados no descritor de processo:

- 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.

Um processo quase sempre faz parte de alguma fila. Em geral, os próprios descritores de processo são utilizados como elementos dessas filas. Antes de ser criado, o descritor do processo faz parte de uma fila de descritores livres. Após a criação, o seu descritor é colocado na fila de aptos. Normalmente, essa fila é mantida na ordem em que os processos deverão receber o processador. O primeiro descritor da fila corresponde ao processo em execução. Ao fazer uma chamada de sistema associada com uma operação de E/S, o descritor do processo em execução é retirado da fila de aptos e inserido na fila associada ao periférico. O contexto de execução do processo é salvo no seu próprio descritor. Após a conclusão da operação de E/S, o seu descritor volta para a fila de aptos. Quando o processo é destruído, o descritor volta para a fila de descritores livres.

Na prática, descritores não são copiados. Todas as filas são implementadas como listas encadeadas. A passagem do descritor de uma fila para a outra é feita através da manipulação de apontadores.

Em muitos sistemas, existe um número fixo de descritores. Ele corresponde ao número máximo de processos que podem existir no sistema. Outros sistemas utilizam alocação dinâmica de memória para criar descritores. Nesse caso, não existe um limite para o número de descritores de processos. As próprias características físicas do equipamento, tais como tamanho da memória principal e velocidade do processador, irão estabelecer limites quanto ao número máximo de processos. Quando alocação dinâmica de memória é utilizada na criação de blocos descritores de processos, é importante que a memória alocada fique dentro da área protegida do sistema operacional. Os descritores de processos contêm informações vitais para a operação do sistema. Em hipótese alguma, eles podem ser alterados por um processo de usuário.

Abaixo está um exemplo de descritor de processo para uma máquina hipotética simples. Suponha que o computador sendo multiprogramado possui os seguintes registradores:

- Apontador de instruções (PC);
- Apontador de pilha (SP);
- Acumulador (ACC);
- Registrador indexador (RX).

Foi utilizada uma estrutura da linguagem C para implementar o bloco descritor. No exemplo, o sistema suporta um número máximo de processos, definido pela constante MAX_DESC_PROC. Os campos relacionados com o uso da memória e de arquivos foram simplificados. É claro que cada sistema operacional utiliza um descritor apropriado para os seus mecanismos de gerência de memória, sistema de arquivos, etc.

```
struct desc_proc{
      char     estado_atual;              /* Estado atual do processo */
      int      prioridade;                /* Prioridade do processo */
      unsigned inicio_memoria;            /* Endereço inicial na memória */
      unsigned tamanho_memoria;           /* Bytes de memória ocupados */
      struct   arquivo arq_abertos[20];   /* Arquivos abertos */

      unsigned tempo_de_cpu;              /* Tempo já gasto de cpu */
      unsigned proc_pc;                   /* Valor salvo do reg. PC */
      unsigned proc_sp;                   /* Valor salvo do reg. SP */
      unsigned proc_acc;                  /* Valor salvo do reg. ACC */
      unsigned proc_rx;                   /* Valor salvo do reg. RX */
      struct   desc_proc *proximo;        /* Aponta para o próximo */
}
```



```
struct desc_proc tab_desc[MAX_DESC_PROC];
struct desc_proc *desc_livre;             /* Lista de descr. livres */
struct desc_proc *espera_cpu;             /* Lista de proc. esperando */
struct desc_proc *usando_cpu;             /* Aponta proc. executando */
```

Na inicialização do sistema, todos os descritores de processo podem ser encadeados, para formar uma "lista de descritores livres". O início dessa lista é apontado por "desc_livre". O código abaixo mostra como ela pode ser inicializada.

```
for(i=0; i<MAX_DESC_PROC - 1; ++i)
      tab_desc[i].proximo = &tab_desc[i+1];

tab_desc[i].proximo = NULL;
desc_livre = &tab_desc[0];
```

O apontador "espera_cpu" indica o início da lista de processos que esperam pelo processador. O descritor do processo que está executando é mantido à parte, apontado por "usando_cpu". Outra solução é fazer com que o processo que está com o processador ocupe a primeira posição da lista apontada por "espera_cpu". Nesse caso, o apontador "usando_cpu" não é necessário.

Para criar um processo, um descritor é retirado da lista apontada por "desc_livre". Se "desc_livre" contém NULL, a "lista de descritores livres" está vazia. Nesse caso, o processo não poderá ser criado. 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.

Quando todos os campos estiverem preenchidos, o descritor do processo é inserido na "lista de espera pelo processador", apontada por "espera_cpu". A partir desse momento, o processo passa a disputar tempo de processador, junto com os demais. Em outras palavras, o processo foi criado.

O procedimento de criação de processo que foi descrito é uma simplificação do que acontece em sistemas reais. Na prática, fatores como a arquitetura do computador e a forma como os diversos módulos do sistema operacional relacionam-se determinam o procedimento exato a ser adotado. Entretanto, o procedimento descrito deixa claro que a criação de um processo, com respeito a gerência do processador, corresponde a uma manipulação de registros e listas encadeadas.

As operações necessárias para a destruição de um processo são semelhantes. Primeiramente, todos os recursos que o processo havia alocado devem ser liberados. Por exemplo, memória principal, impressora, arquivos, etc. Essa etapa corresponde à limpeza do ambiente após a morte do processo. Depois, o seu descritor de processo é retirado da lista em que está e inserido novamente na "lista de descritores livres". Nesse momento, o processo deixa de existir. O descritor será reaproveitado na criação de um novo processo.

No sistema descrito, o conceito de grupo de processos pode ser implementado através de um campo adicional no descritor do processo. Esse campo indica qual é o grupo do processo. Para realizar uma operação do tipo "elimina todos os processos de um determinado grupo", é feita uma pesquisa seqüencial sobre todos os descritores de processo em uso. Os procedimentos de destruição de processo são repetidos para todos aqueles que forem do grupo especificado.

Uma solução alternativa é formar uma lista encadeada para cada grupo de processos. Um novo campo semelhante ao campo "proximo" é definido. Cada descritor de processo passa a ser elemento

de duas listas encadeadas: a lista de recurso (desc_livre, espera_cpu, etc) e a lista que forma o seu grupo. Essa solução dispensa a pesquisa seqüencial sobre todos os descritores de processo do sistema.

4.2 Chaveamento de contexto

A base da multiprogramação é o compartilhamento (multiplexação) do processador entre os processos. Por exemplo, enquanto o processo 1 fica bloqueado, à espera de um dispositivo periférico mais lento, o processo 2 ocupa o processador. 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**. Para passar o processador do processo 1 para o processo 2, é necessário salvar o contexto de execução do processo 1. Quando o processo 1 receber novamente o processador, o seu contexto de execução será restaurado. É necessário salvar tudo que poderá ser destruído pelo processo 2, enquanto ele executa.

O **contexto de execução** é formado basicamente pelos registradores do processador. O processo 2, ao executar, vai colocar seus próprios valores nos registradores. Entretanto, quando o processo 1 voltar a executar, ele espera encontrar nos registradores os mesmos valores que havia no momento da interrupção. O programa do usuário sequer sabe que será interrompido diversas vezes durante a sua execução. Logo, não é possível deixar para o programa a tarefa de salvar os registradores. Isso deve ser feito pelo próprio sistema operacional.

Os conteúdos dos registradores são salvos toda vez que um processo perde o processador. Eles são recolocados quando o processo volta a executar. Dessa forma, o processo não percebe que foi interrompido. Em geral, salvar o contexto de execução do processo em execução é a primeira tarefa do sistema operacional, ao ser acionado. Da mesma forma, a última tarefa do sistema operacional ao entregar o processador para um processo é repor o seu contexto de execução. Ao repor o valor usado pelo processo no apontador de instruções (*program counter*), o processador volta a executar instruções do programa do usuário. O módulo do sistema operacional que realiza a reposição do contexto é chamado de *dispatcher*.

O local usado para salvar o contexto de execução de um processo é o seu próprio bloco descritor. Uma solução alternativa é salvar todos os registradores na própria pilha do processo, colocando no bloco descritor apenas um apontador para o topo da pilha. Essa solução é, em geral, mais simples e eficiente. O próprio mecanismo de atendimento de interrupções da maioria dos processadores já salva na pilha alguns registradores. Entretanto, essa solução utiliza uma pilha cujo controle de espaço disponível está a cargo do usuário. Se o salvamento de contexto ocorrer no momento em que não existe espaço suficiente na pilha, informações pertencentes ao espaço de endereçamento do usuário poderão ser destruídas. Essa solução somente é viável quando é possível garantir que haverá espaço disponível na pilha do usuário. Uma solução de compromisso é fazer com que cada processo possua uma pilha adicional para uso exclusivo do sistema operacional. Nesse caso, primeiramente é feito um chaveamento de pilha. Depois, o contexto é salvo na pilha do processo, controlada pelo sistema operacional. Nessa pilha, é possível garantir que haverá espaço suficiente para o contexto de execução do processo.

Pode-se ilustrar a tarefa de reposição do contexto de execução de um processo através da máquina hipotética anteriormente apresentada (seção 4.1). Vamos supor que o contexto foi salvo no próprio bloco descritor. O endereço do bloco descritor do processo, contido na variável "usando_cpu", é copiado para o registrador RX. A partir desse ponto, utilizando deslocamentos apropriados, é possível acessar os campos do descritor que contêm o contexto de execução. Esses valores são

copiados para os registradores do processador. Quando o registrador PC for alterado, o processador voltará a executar o programa associado ao processo em questão. O código abaixo ilustra como essa tarefa poderia ser feita. Nas instruções de MOVE, o operando origem é o primeiro da instrução. Os símbolos utilizados são os mesmos do código em C mostrado na seção anterior.

```
MOVE    USANDO_CPU, RX          ;Faz RX apontar descritor
MOVE    [RX+PROC_SP], SP        ;Copia campo proc_sp p/ sp
MOVE    [RX+PROC_PC], ACC       ;Pega campo proc_pc
PUSH    ACC                     ;Coloca no topo da pilha
MOVE    [RX+PROC_ACC], ACC      ;Copia campo proc_acc p/acc
MOVE    [RX+PROC_RX], RX        ;Copia campo proc_rx p/rx
RET                             ;Coloca topo da pilha em pc
```

Como o conteúdo do campo "proc_pc" do descritor do processo foi colocado na pilha, ao executar um retorno de sub-rotina o efeito será o mesmo de mover "proc_pc" para o registrador PC. A forma exata a ser utilizada para repor o contexto depende da arquitetura em questão. Muitos computadores oferecem instruções específicas que facilitam essa operação.

4.3 Threads

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, sincronização) é "mais leve" quando comparada com processos. Por exemplo, criar um processo implica alocar e inicializar estruturas de dados no sistema operacional para representá-lo. Por outro lado, criar uma *thread* implica apenas definir uma pilha e um novo contexto de execução dentro de um processo já existente. O chaveamento entre duas *threads* de um mesmo processo é muito mais rápido que o chaveamento entre dois processos. Por exemplo, como todas as *threads* de uma mesmo processo compartilham o mesmo espaço de endereçamento, a MMU (*memory management unit*) não é afetada pelo chaveamento entre elas. Em função do exposto acima, *threads* são muitas vezes chamadas de **processos leves**.

Duas maneiras básicas podem ser utilizadas para implementar o conceito de *threads* em um sistema. Na primeira, o sistema operacional suporta apenas processos convencionais, isto é, processos com uma única *thread*. O conceito de *thread* é então implementado pelo próprio processo a partir de uma biblioteca ligada ao programa do usuário. Devido a essa característica, *threads* implementadas dessa forma são denominadas de ***threads* do nível do usuário** (*user-level threads*). No segundo caso, o sistema operacional suporta diretamente o conceito de *thread*. A gerência de fluxos de execução pelo sistema operacional não é mais orientada a processos mas sim a *threads*. As *threads* que seguem esse modelo são ditas ***threads* do nível do sistema** (*kernel threads*).

O primeiro método é denominado N:1 (*many-to-one*). A principal vantagem é o fato de as *threads* serem implementadas em espaço de usuário, não exigindo assim nenhuma interação com o sistema

operacional. Esse tipo de *thread* oferece um chaveamento de contexto mais rápido e menor custo para criação e destruição. A biblioteca de *threads* é responsável pelo compartilhamento, entre elas, do tempo alocado ao processo. O sistema operacional preocupa-se apenas em dividir o tempo do processador entre os diferentes processos. A grande desvantagem desse método é que as *threads* são efetivamente simuladas a partir de um único fluxo de execução pertencente a um processo convencional. Como conseqüência, qualquer paralelismo real disponível no computador não pode ser aproveitado pelo programa, embora permita vários processos diferentes executarem ao mesmo tempo. Outra conseqüência é que uma *thread* efetuando uma operação de entrada ou saída bloqueante provoca o bloqueio de todas as *threads* do seu processo. Existem técnicas de programação para evitar isso, mas são relativamente complexas.

O segundo método é dito 1:1 (*one-to-one*). Ele resolve os dois problemas mencionados acima: aproveitamento do paralelismo real dentro de um único programa e processamento junto com E/S. Para que isso seja possível, o sistema operacional deve ser projetado de forma a considerar a existência de *threads* dividindo o espaço de endereçamento do processo hospedeiro. A desvantagem desse método é que as operações relacionadas com as *threads* passam necessariamente por chamadas ao sistema operacional, o que torna *threads* tipo 1:1 "menos leves" que *threads* do tipo N:1.

Existe um método misto que tenta combinar as duas abordagens, chamado M:N (*many-to-many*). Esse método possui escalonamento nos dois níveis. Uma biblioteca no espaço do usuário seleciona *threads* (M) do programa para serem executadas em uma ou mais *threads* do sistema (N). Embora flexível, esse método exige que o programador decida qual o valor de N e como será a execução das M *threads* do programa pelas N *threads* do sistema.

4.4 Escalonadores

Em qualquer sistema operacional que implemente multiprogramação, diversos processos disputam os recursos disponíveis no sistema, a cada momento. Logo, existe a necessidade de escalonar esses processos com respeito à utilização dos recursos. Em particular, é necessário dividir o recurso "tempo do processador" entre os processos do sistema. Isso é feito de duas maneiras distintas: escalonamento dos processos a curto prazo e escalonamento dos processos a longo prazo.

No **escalonamento de curto prazo** é decidido qual processo será executado a seguir. Esse escalonador é executado com grande freqüência, sempre que o processador ficar livre. Logo, deve ser rápido.

Em alguns sistemas, processos nem sempre são criados no momento da solicitação. Em alguns ambientes, a criação de um processo pode ser postergada se a carga na máquina estiver muito grande. Cabe ao **escalonador de longo prazo** decidir quando um processo solicitado será efetivamente criado. Pode-se esperar a carga na máquina diminuir para então disparar um novo processo. Em geral, esse tipo de escalonador não é utilizado em sistemas nos quais um usuário aguarda no terminal pelo início da execução do processo. O processo é sempre criado, mesmo que o aumento na carga da máquina piore o tempo de resposta dos processos.

É sempre possível que uma sobrecarga no sistema leve ao esgotamento dos recursos disponíveis. Em especial, isso pode ocorrer com a memória principal. A memória necessária para os processos em execução pode tornar-se maior que o total de memória disponível. Mesmo que isso não seja verdade no momento da criação dos processos, muitos programas alocam memória dinamicamente, durante a sua execução. Uma solução, nesse caso, é a técnica conhecida como ***swapping***.

Por exemplo, considere um sistema no qual existem 10 processos, mas somente 8 deles cabem na memória principal. A cada momento, 2 processos são completamente copiados para disco. Com o passar do tempo, os processos revezam-se no disco. Dessa maneira, os 10 são executados em uma memória que só comporta 8 deles.

Na operação de ***swap-out***, a execução de um processo é temporariamente suspensa, e o seu código e dados são copiados para o disco. A operação ***swap-in*** faz o contrário. O processo é copiado de volta do disco para a memória e sua execução é retomada do ponto em que havia sido suspensa. Enquanto está suspenso, o processo não disputa tempo do processador, nem ocupa memória principal. O escalonador que decide qual processo deverá sofrer *swap-in* ou *swap-out* é chamado **escalonador de médio prazo**. Esse escalonador está ligado tanto à gerência de memória quanto à gerência do processador.

O escalonador mais importante é o de curto prazo. Em geral, quando o termo ***scheduler*** é empregado sem nenhum complemento, refere-se ao escalonador de curto prazo. Embora as técnicas de *swapping* (médio prazo) e controle da carga na criação dos processos (longo prazo) sejam opcionais, o escalonamento do processador é indispensável na implementação do conceito de processo.

4.5 Algoritmos de escalonamento

Nesta seção serão vistos algoritmos para o escalonador de curto prazo. Em geral, esses mesmos algoritmos podem ser facilmente adaptados para a situação de médio e longo prazo. Para a apresentação dos algoritmos, é suposto que existem no sistema diversos processos prontos para receber o processador. A tarefa do algoritmo é justamente escolher qual deles será executado a seguir.

Na escolha de um algoritmo de escalonamento, utiliza-se como critério básico o objetivo de aumentar a produção do sistema e, ao mesmo tempo, diminuir o tempo de resposta percebido pelos usuários. Esses dois objetivos podem tornar-se conflitantes em determinadas situações.

Para aumentar a produção do sistema (*throughput*), é necessário manter o processador ocupado todo o tempo. Dessa forma, o sistema produz mais em menos tempo. Também é importante oferecer um baixo tempo de resposta (*turnaround time*) ao usuário. Isso é obtido, no caso da gerência do processador, com um baixo tempo médio de espera na fila do processador.

Para todos os algoritmos, é necessário considerar a existência de processos "*i/o-bound*" e "*cpu-bound*" misturados no sistema. Também não adianta um baixo tempo médio de resposta com variância elevada. 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.

4.5.1 Ordem de chegada (FIFO - *First-in first-out*)

Esse é o algoritmo de implementação mais simples. A fila do processador é uma fila simples. Os processos são executados na mesma ordem em que chegaram na fila. Um processo somente libera o processador quando realiza uma chamada de sistema ou quando ocorre algum erro na execução. As chamadas de sistema incluem a própria indicação de término do processo.

O problema desse algoritmo é o desempenho. Quando um processo "*cpu-bound*" está na frente da fila, todos os processos devem esperar que ele termine seu ciclo de processador para então executar.

Por exemplo, a tabela abaixo mostra uma fila com 4 processos. Para cada processo, está indicada a duração do seu próximo ciclo de processador em unidades de tempo.

Processo	Duração do próximo ciclo de processador	
A	12	(Unidades de tempo)
B	8	"
C	15	"
D	5	"

Supondo que a ordem dos processos na tabela é a própria ordem da fila, o diagrama na Figura 4.1 mostra a sua seqüência de execução. O tempo médio de espera na fila do processador para esse conjunto de processos foi de (0+12+20+35)÷4=67÷4=16.75. Obviamente, se o processo "D" fosse executado antes do processo "C", o tempo médio de espera na fila seria melhor. Esse fato justifica a proposição do próximo algoritmo.

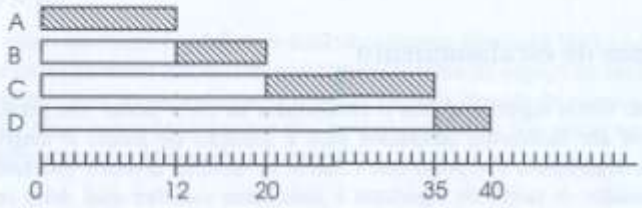

Figura 4.1 – Diagrama de tempo usando FIFO.

4.5.2 Ciclo de processador menor antes (SJF - *Shortest job first*)

O menor tempo médio de espera na fila é obtido quando é selecionado antes o processo cujo próximo ciclo de processador é o menor entre os processos que estão na fila. Esse algoritmo poderia ser implementado como uma lista ordenada na ordem crescente da duração do próximo ciclo de processador. Os processos são sempre inseridos na lista de maneira a manter a ordenação proposta. No momento de escolher um processo para executar, basta pegar o primeiro da lista.

Utilizando a mesma tabela de processos da seção anterior, o diagrama de tempo de execução ficaria como mostrado na Figura 4.2. Agora o tempo médio de espera na fila do processador, para esse conjunto de processos, foi de (0+5+13+25)÷4=43÷4=10.75.O mesmo trabalho foi feito no mesmo tempo mas, em média, os processos (usuários) tiveram que esperar menos por isso. O problema desse algoritmo é que, para implementá-lo, é necessário prever o futuro. A duração do próximo ciclo de processador de um processo não é conhecida. Ela depende, entre outras coisas, dos dados de entrada do programa em execução.

Mesmo não podendo ser implementado, esse algoritmo é útil por duas razões. Primeiramente, ele oferece um limite teórico para o tempo médio de espera. Esse algoritmo pode ser comparado com outros, implementáveis, em simulações. Pode-se também implementar aproximações do SJF nas quais a duração dos ciclos de processador mais recentes são utilizadas como uma aproximação para os próximos ciclos. Nesse caso, é suposto que o comportamento do processo não mudará significativamente com respeito à duração dos ciclos de processador.

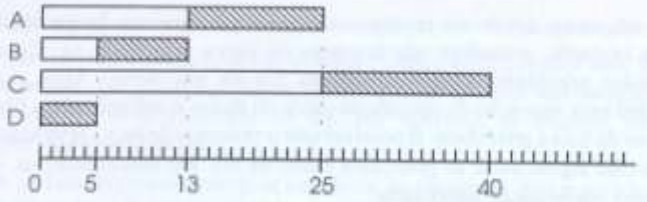

Figura 4.2 – Diagrama de tempo usando SJF.

É importante salientar que, nesse algoritmo, os processos "*i/o-bound*" são favorecidos. Isso acontece porque eles recebem e liberam o processador rapidamente, minimizando o tempo de espera dos demais. Como regra geral, favorecer os processos "*i/o-bound*" significa diminuir o tempo médio de espera na fila do processador.

4.5.3 Prioridade

Quando os processos de um sistema possuem diferentes prioridades, essa prioridade pode ser utilizada para decidir qual processo é executado a seguir. Um algoritmo de escalonamento desse tipo pode ser implementado através de uma lista ordenada conforme a prioridade dos processos. O processo a ser executado é sempre o primeiro da fila. Quando dois processos possuem a mesma prioridade, algum critério para desempate deve ser utilizado. Por exemplo, a ordem de chegada na fila do processador (FIFO).

A prioridade de um processo pode ser definida externamente ao sistema. Por exemplo, a administração do Centro de Processamento de Dados confere prioridade maior aos processos disparados pelos seus funcionários e prioridade menor aos processos disparados por alunos. Também é possível que o próprio sistema defina a prioridade dos processos. Por exemplo, se os processos "*i/o-bound*" possuírem prioridade maior, o algoritmo irá aproximar-se de SJF.

A tabela abaixo descreve uma fila de processador hipotética. Para cada processo, foi especificada uma prioridade. Nesse sistema, quanto menor o valor numérico da sua prioridade, mais importante é o processo. Na Figura 4.3 está o diagrama de tempo da execução.

Processo	Prioridade	Duração do próximo ciclo de processador	
A	3	12	(Unidades de tempo)
B	4	8	"
C	2	15	"
D	1	5	"

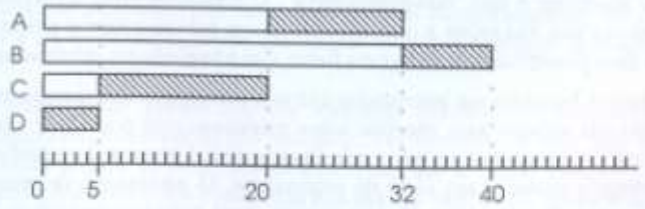

Figura 4.3 – Diagrama de tempo usando prioridades.

Vários aspectos adicionais devem ser considerados quando o algoritmo de escalonamento emprega prioridades. Por exemplo, considere um processo de baixa prioridade na fila do processador. Processos de maior prioridade sempre entram na fila na sua frente. Após executarem, esses processos realizam uma operação de entrada ou saída de dados e voltam para a fila, novamente na frente do processo de baixa prioridade. É possível que o processo de baixa prioridade nunca execute porque sempre existe algum outro de prioridade maior na fila. Em outras palavras, é possível que o processo sofra uma **postergação indefinida**.

É impossível ocorrer postergação indefinida quando o algoritmo é FIFO. Uma vez na fila do processador, é garantido que chegará a vez de o processo ser executado. Já em SJF um processo com a perspectiva de um grande ciclo de processador poderá ser eternamente preterido em favor de processos com ciclos de processador pequenos. Logo, também é possível ocorrer postergação indefinida quando o algoritmo é SJF.

Pode-se impedir a ocorrência de postergação indefinida quando prioridades são utilizadas, através da adição de um algoritmo de envelhecimento (*aging*) ao método básico. Lentamente, os processos na fila têm a sua prioridade elevada. Após algum tempo, processos que envelheceram sem executar tem a prioridade elevada de tal modo que finalmente conseguem obter o processador. Após o ciclo de processador e o ciclo de E/S, esse processo volta para a fila com a sua prioridade original. O objetivo do envelhecimento não é elevar a prioridade do processo, mas impedir que ele fique na fila para sempre. Uma vez que o processo conseguiu executar um ciclo, deve passar novamente pelo mecanismo de envelhecimento. Afinal, se o processo tem uma prioridade baixa, é natural que ele execute lentamente.

Considere agora um outro 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. Nesse caso, temos "prioridade não-preemptiva" como algoritmo de escalonamento.
- 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".

Em resumo, um algoritmo é dito "**preemptivo**" se o processo em execução puder perder o processador para outro processo, por algum motivo que não seja o término do seu ciclo de processador. Se o processo em execução só libera o processador por vontade própria (chamada de sistema), então o algoritmo é dito "**não-preemptivo**". O algoritmo FIFO é intrinsecamente "não-preemptivo", uma vez que não existe a possibilidade de um processo tirar o processador de outro. Já SJF admite as duas possibilidades, da mesma forma que o baseado em prioridades.

Tipicamente, soluções baseadas em prioridades utilizam preempção. Em termos práticos não tem sentido fazer todo um esforço para executar antes processos com prioridade alta e, ao mesmo tempo, permitir que um processo com baixa prioridade ocupe o processador indefinidamente, uma vez que ele conseguiu iniciar o seu ciclo de processador. O mecanismo de preempção permite implementar o conceito de prioridades de uma maneira mais completa no sistema, sendo por isso normalmente utilizado.

4.5.4 Fatia de tempo

Nesse método, também conhecido como ***round-robin***, cada processo recebe uma fatia de tempo do processador (***quantum***). Ele pode ser implementado através de uma fila simples, semelhante ao FIFO. Processos entram sempre no fim da fila. No momento de escolher um processo para executar, é sempre o primeiro da fila.

A diferença está no fato do processo receber uma **fatia de tempo** ao iniciar o ciclo de processador. 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. O novo primeiro processo da fila inicia então sua fatia.

Utilizando a mesma tabela de processos das seções anteriores e uma fatia de três unidades de tempo, o diagrama ficaria como mostrado na Figura 4.4.

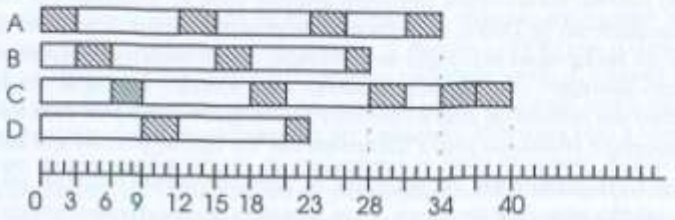

Figura 4.4 – Diagrama de tempo usando fatias de tempo.

Um relógio de tempo real em hardware delimita as fatias de tempo através de interrupções. Nesse algoritmo não é possível postergação indefinida, pois processos sempre entram no fim da fila. Não existe como um processo ser "passado para trás".

Um problema é definir o tamanho da fatia de tempo. É preciso levar em conta que o chaveamento entre processos não é instantâneo, como suposto no diagrama. Ele envolve um custo em termos de tempo do processador. Se a fatia de tempo for muito pequena, esse custo cresce em termos relativos. Por exemplo, vamos supor que, em determinado equipamento, o tempo médio de execução para uma instrução de máquina é 1µs (um microsegundo). Vamos também supor que são necessárias 200 instruções de máquina para realizar o chaveamento entre dois processos (salvamento de contexto, manipulação da fila do processador, carga de contexto). Nesse caso, o chaveamento de contexto consome 200µs (200 × 1µs) do tempo do processador. Se a fatia de tempo escolhida for 1ms (um milisegundo), então 20% do tempo do processador será gasto apenas efetuando trocas de contexto entre processos. Ninguém compra um computador para que ele fique chaveando contexto de processos. Um valor mais razoável economicamente seria algo menor que 1%. Para chegar a 1% no exemplo, temos que utilizar uma fatia de tempo na ordem de 20ms (vinte milisegundos).

Uma fatia de tempo muito grande também apresenta problemas. Principalmente, perde-se a aparência de paralelismo na execução dos processos. Por exemplo, se uma fatia de 1 segundo for utilizada, e o sistema possuir 20 processos em execução, cada processo receberá 1s a cada 20s. No terminal, o usuário vai perceber que o programa executa aos "pulinhos", ao invés de apresentar uma execução em velocidade mais ou menos constante. Uma fatia de tempo de 100ms no mesmo

sistema significaria o processo receber 0.1s a cada 2s. O fenômeno dos "pulinhos" já não seria tão gritante.

O exemplo dos 20 processos serve para ilustrar a importância da fatia de tempo na ilusão de paralelismo criada pela multiprogramação. Entretanto, dificilmente na prática um processo executa durante um segundo completo sem fazer algum tipo de chamada de sistema e ficar bloqueado durante a sua realização. Isso só vai acontecer com processos executando programas que utilizam intensamente o processador, como na área de cálculo numérico e computação gráfica.

Quando a fatia de tempo é tão grande que todos os processos liberam o processador (fazem uma chamada de sistema) antes dela terminar, o algoritmo degrada para FIFO. Por outro lado, em sistemas onde a ilusão de paralelismo não é tão importante, pode-se aumentar a fatia de tempo para reduzir o custo associado com chaveamento de processos.

4.5.5 Múltiplas filas

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 todo o tempo (execução em ***foreground***). É possível construir um sistema no qual existe uma fila de processador para cada tipo de processo. Quando um processo é criado, vai para a fila apropriada. É o tipo do processo que define a sua fila.

No exemplo do CPD citado antes, os processos do desenvolvimento poderiam ser colocados em uma fila que trabalha com fatia de tempo. Já os processos em *background* seriam enviados para uma outra fila, escalonando conforme a ordem de chegada (FIFO). Ainda é necessário um terceiro algoritmo, para definir qual fila ocupa o processador a cada momento. Por exemplo, prioridade preemptiva pode ser utilizada. A fila *foreground* tem maior prioridade, enquanto a fila *background* só ganha o processador (seus processos somente são executados) quando a outra fila estiver vazia. Dessa forma, durante o dia o computador atende principalmente o desenvolvimento, executando programas da produção quando possível. Durante a madrugada, o desenvolvimento pára, permitindo então que os processos em *background* tomem conta da máquina.

Uma alternativa para o algoritmo entre filas apresentado seria utilizar também fatias de tempo para as filas. Por exemplo, a fila *foreground* recebe 60% do tempo do processador enquanto a fila *background* recebe 40% do tempo. Obviamente, nos momentos em que uma das filas estiver vazia, a outra ocupa o processador todo o tempo.

Um algoritmo com múltiplas filas bastante popular é utilizar fatia de tempo dentro das filas e prioridade preemptiva entre elas. Dentro de uma mesma prioridade, processos dividem o tempo do processador em fatias. Porém, o algoritmo permite favorecer determinados processos, concedendo a esses uma prioridade elevada. Isso é feito normalmente para os processos que executam tarefas para o próprio sistema operacional. Na verdade, esse algoritmo não precisa realmente ser implementado através de várias filas. Ele pode ser implementado através de uma fila única, onde os processos são mantidos ordenados na ordem decrescente das prioridade. Na seleção, é sempre escolhido o primeiro processo da lista. Na inserção, o processo é sempre colocado após todos os processos de prioridade superior ou igual. Para processos de mesma prioridade, é obtido o efeito circular necessário para o algoritmo baseado em fatias de tempo.

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**.

Por exemplo, pode-se utilizar esse tipo de fila para construir um algoritmo que favorece os processos "*i/o-bound*". Para tanto, são utilizadas duas filas, A e B, cada uma trabalhando com fatias de tempo. Prioridade preemptiva é utilizada entre filas, sendo que a fila A possui prioridade maior. Quando um processo é criado, ou quando volta de uma chamada de sistema, sempre é inserido na fila A. Entretanto, se o processo esgota uma fatia de tempo da fila A, ele é imediatamente transferido para a fila B.

No algoritmo apresentado, processos "*i/o-bound*" em geral permanecem na fila A, pois fazem uma chamada de sistema antes de esgotar a sua fatia de tempo. Já processos "*cpu-bound*" esgotam a fatia de tempo e são transferidos para a fila B. Permanecendo o tempo todo na fila A, processos "*i/o-bound*" acabam recebendo um serviço melhor no que diz respeito ao tempo do processador. O sistema apresenta um "efeito peneira", onde processos "*cpu-bound*" acabam descendo para a fila B. Os mesmos processos "*cpu-bound*" são inseridos na fila A quando voltam da chamada de sistema. Isso significa que, mesmo se um processo mudar seu comportamento durante a execução, ele receberá um tratamento adequado do sistema.

Outro exemplo desse tipo de algoritmo de escalonamento é utilizado em versões tradicionais do sistema operacional UNIX, tais como o Unix System V release 3 (SVR3) ou o Berkeley Software Distribution 4.3 (4.3BSD). O Unix tradicional emprega prioridade variável. Quando um processo é liberado e possui prioridade maior do que o processo que está executando, existe um chaveamento de contexto, e o processo recém liberado passa a ser executado. Processos com a mesma prioridade dividem o tempo do processador através do mecanismo de fatias de tempo. A prioridade de cada processo varia conforme o seu padrão de uso do processador. Além disso, um processo executando código do sistema operacional não pode ser preemptado. O livro [VAH96] descreve as soluções empregadas em diversos sistemas operacionais tipo Unix, inclusive Solaris e Mach.

No UNIX SVR3, as prioridades variam entre 0 e 127, onde um número menor representa prioridade mais alta. Os valores entre 0 e 49 são reservados para processos executando código do kernel (núcleo). Os valores entre 50 e 127 são para processos em modo usuário.

O descritor de processo contém, entre outras informações:

- A prioridade atual do processo, `p_pri`;
- A prioridade desse processo quando em modo usuário, `p_usrpri`;
- Uma medida da utilização recente de processador por esse processo, `p_cpu`;
- Um fator de gentileza definido pelo programador ou administrador do sistema, `p_nice`;

Quando em modo usuário, o processo possui sua prioridade definida por `p_usrpri`, isto é, `p_pri=p_usrpri`. Quando um processo é liberado dentro do kernel após ter acordado de um bloqueio, ele recebe um "empurrão temporário", na forma de uma prioridade `p_pri` que é numericamente menor do que o seu `p_usrpri`. Cada razão de bloqueio tem uma "*sleep priority*" associada, a qual determina a "força do empurrão". Por exemplo, a prioridade após ficar esperando por entrada de terminal é 28, e a prioridade após ficar esperando por um acesso ao disco é 20. Quando um processo acorda, `p_pri` recebe a "*sleep priority*" correspondente ao bloqueio. A prioridade do processo retornará para o valor `p_usrpri` quando esse voltar para modo usuário.

O valor de `p_usrpri` depende dos valores de `p_cpu` e de `p_nice` do processo em questão. O fator de gentileza é um número entre 0 e 39, cujo valor padrão (*default*) é 20. O valor de `p_cpu` inicial é zero na criação do processo. A cada interrupção do temporizador (*tick*), o valor `p_cpu` do processo em execução naquele instante é incrementado, até um valor máximo de 127.

Simultaneamente, a cada segundo, os valores `p_cpu` de todos os processos são reduzidos por um fator de decaimento (*decay factor*). Por exemplo, são multiplicados por 1/2, ou são multiplicados por `decay`, onde `decay=(2*load_average)/(2*load_average+1)`, e o valor `load_average` é o número médio de processos aptos a executar dentro do último segundo.

A prioridade do processo quando executando código da aplicação é calculada através da fórmula: `p_usrpri=50+(p_cpu/4)+(2*p_nice)`. Em função desse recálculo, pode haver um chaveamento de contexto. Isso acontece quando o processo em execução fica com prioridade mais baixa do que qualquer outro processo apto a executar, considerando-se os novos valores de `p_usrpri` de todos os processos.

4.5.6 Considerações finais

A partir dos algoritmos básicos apresentados, é possível imaginar dezenas de combinações e variações. Tipicamente, a maioria dos sistemas trabalham com fatia de tempo. Também é usual empregarem prioridade para favorecer determinados processos que realizam tarefas para o próprio sistema operacional. Nesses sistemas, a ilusão de paralelismo é fundamental. Uma implementação simples para IBM-PC é descrita em [BIG86].

Para processos em *background*, quando não existe um usuário esperando que a resposta apareça na tela, é natural utilizar uma fatia de tempo maior. Até mesmo FIFO é viável, desde que com proteção contra laços infinitos em programas (tempo máximo de execução após o que o processo é abortado).

Por outro lado, em sistemas de tempo real que atendem a eventos externos, tais como fornos, motores e válvulas, prioridade nesse caso é essencial. Esses processos precisam atender a eventos externos dentro de limites bem definidos de tempo. Sistemas operacionais de tempo real devem apresentar características especiais no que diz respeito ao tempo de execução dos processos. O atendimento desses requisitos especiais exige o emprego de algoritmos de escalonamento apropriados. Escalonamento de processos para sistemas de tempo real são discutidos em [FAR00].

4.6 Exercícios

1) Explique quando são necessários e quais as funções dos escalonadores de curto termo, médio termo e longo termo. (Seção 4.4)

2) Mostre através de um exemplo a razão de ser genericamente melhor favorecer os processos *i/o-bound* na disputa pelo processador. (Seção 4.5)

3) Crie um algoritmo de escalonamento baseado em múltiplas filas com realimentação. Devem existir duas filas. O algoritmo entre filas deve trabalhar de forma que, com o passar do tempo, processos "*i/o-bound*" vão para a fila 1, e processos "*cpu-bound*", para a fila 2. Não deve ser possível a ocorrência de postergação indefinida de nenhum processo. (Seção 4.5)

4) Que tipos de critérios devem ser utilizados no momento da definição da fatia de tempo a ser empregada em um determinado sistema? (Seção 4.5)

5) Em um sistema operacional, o escalonador de curto prazo utiliza duas filas. A fila "A" contém os processos do pessoal do CPD e a fila "B" contém os processos dos alunos. O algoritmo entre filas é fatia de tempo. De cada 11 unidades de tempo de processador, 7 são fornecidas para os processos da fila "A", e 4 para os processos da fila "B". O tempo de cada fila é dividido entre os processos também por fatias de tempo, com fatias de 2 unidades para todos. A tabela abaixo mostra o

conteúdo das duas filas no instante zero. Considere que está iniciando um ciclo de 11 unidades, e agora a fila "A" vai receber as suas 7 unidades de tempo. Mostre a seqüência de execução dos processos, com os momentos em que é feita a troca. (Seção 4.5)

OBS: Se terminar a fatia de tempo da fila "X" no meio da fatia de tempo de um dos processos, o processador passa para a outra fila. Entretanto, esse processo permanece como primeiro da fila "X", até que toda sua fatia de tempo seja consumida.

Fila	Processo	Duração do próximo ciclo de processador
A	P1	6
A	P2	5
A	P3	7
B	P4	3
B	P5	8
B	P6	4

6) Considerando a fila do processador mostrada abaixo, calcule o instante no qual cada processo conclui o seu ciclo de processador, caso o algoritmo de escalonamento utilizado seja "Fatias de tempo com tamanho 3". Construa o diagrama de Gantt (diagrama de tempo). Observe que alguns processos estão na fila já no instante zero, enquanto outros entram na fila com o algoritmo já em execução. (Seção 4.5)

Processo	Instante de ingresso na fila	Duração do próximo ciclo de processador
A	0	4
B	0	5
C	0	8
D	5	3
E	8	6
F	13	4

7) Quatro programas devem ser executados em um computador. Todos os programas são compostos por 2 ciclos de processador e 2 ciclos de E/S. A entrada e saída de todos os programas é feita sobre a mesma unidade de disco. Os tempos para cada ciclo de cada programa são mostrados abaixo:

Programa	Processador	Disco	Processador	Disco
P1	3	10	3	12
P2	4	12	6	8
P3	7	8	8	10
P4	6	14	2	10

Construa um diagrama de tempo mostrando qual programa está ocupando o processador e o disco a cada momento, até que os 4 programas terminem. Suponha que o algoritmo de escalonamento utilizado seja fatia de tempo, com fatias de 4 unidades. Qual a taxa de ocupação do processador e do disco? (Seção 4.5)

8) O que acontece com as duas taxas de ocupação calculadas no problema anterior se for utilizado um disco com o dobro da velocidade de acesso (duração dos ciclos de E/S é dividida por dois)? (Seção 4.5)

6
Gerência de 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. Como será visto neste capítulo, existem diversas técnicas para a gerência de memória. Ao longo deste capítulo, é suposto que, para executar, um processo precisa estar todo ele na memória principal. Esse requisito será relaxado no capítulo que apresenta o conceito de memória virtual.

A técnica particular que determinado sistema operacional emprega depende, entre outras coisas, de o que a arquitetura do computador em questão suporta. As técnicas de gerência de memória estão intimamente ligadas ao hardware do computador. Em [BAC86], [COM84], [TAN87] e [VAH96], podem ser encontradas as descrições de algumas soluções empregadas em sistemas operacionais específicos. Essas soluções variam com relação à funcionalidade oferecida e à complexidade dos mecanismos empregados. Na prática, as arquiteturas existentes no mercado possuem uma série de detalhes que tornam complexa a implementação dos mecanismos de gerência de memória. Uma excelente descrição de arquiteturas contemporâneas pode ser encontrada em [JAC98a] e [JAC98b].

6.1 Memória lógica e memória física

A **memória lógica** de um processo é aquela que o processo enxerga, ou seja, aquela que o processo é capaz de endereçar e acessar usando as suas instruções. 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 básicos 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.

A Figura 6.1 ilustra o papel da MMU entre o processador e a memória. Na verdade, o processador e a MMU formam, na maioria das vezes, um único circuito integrado. É interessante observar que o processador trabalha sempre com endereços lógicos.

Figura 6.1 - Diagrama incluindo a MMU entre o processador e a memória.

No capítulo sobre multiprogramação, foi mostrado um exemplo de hardware de proteção para a memória. A Figura 6.2 reproduz aquele exemplo. Podemos considerar os dois registradores de limite como uma MMU muito simples. Nesse caso, considera-se ainda que os endereços lógicos e físicos possuem valores idênticos. Isto é, quando o processo gera o endereço lógico 123, a memória recebe o endereço físico 123. Entretanto, o espaço de endereçamento lógico de um processo de usuário é limitado. O conteúdo dos registradores de limite em um dado momento definem o espaço de endereçamento lógico. No exemplo da Figura 6.2, o espaço de endereçamento lógico do processo em execução vai de 100 a 799. Qualquer endereço lógico fora desse intervalo será considerado ilegal. Já o espaço de endereçamento físico é sempre toda a memória principal.

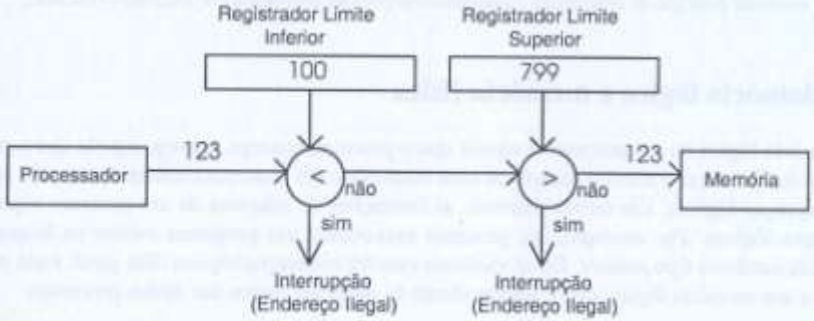

Figura 6.2 - Mecanismo de proteção para a memória com registradores de limite.

Uma outra forma de MMU simples é mostrada na Figura 6.3. Agora o endereço lógico gerado pelo processo é primeiro comparado com um limite superior. Caso seja menor ou igual, ele então é somado ao valor do registrador de base. O resultado da soma é o endereço físico que vai para a memória. Nesse esquema, o endereço lógico é transformado em endereço físico através da soma do valor da base. Temos então o endereço lógico diferente do respectivo endereço físico.

Nesse esquema de MMU, o espaço de endereçamento lógico vai de zero até o valor limite. Esses são os endereços de memória manipulados pelo processo. No caso do exemplo da Figura 6.3, o processo pode gerar endereços lógicos entre zero e 200. Qualquer valor fora desse intervalo será considerado ilegal. Os endereços lógicos são mapeados pela MMU para uma área do espaço de

endereçamento físico. Essa área da memória física inicia no valor indicado pelo registrador de base e tem o mesmo tamanho da memória lógica do processo. Observe que a proteção de memória é conseguida, pois o processo de usuário está restrito a essa área da memória física.

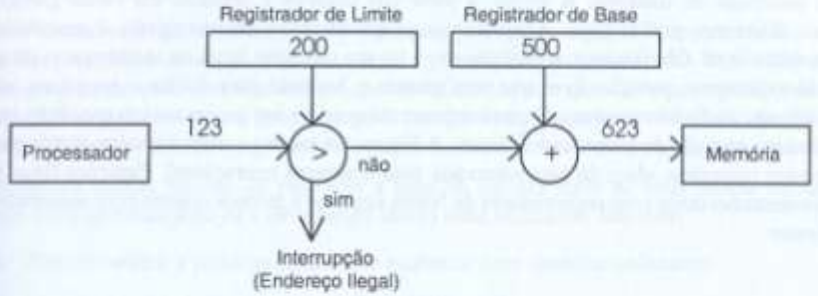

Figura 6.3 - Mecanismo de proteção para a memória com registradores de base e limite.

Tanto os registradores de limite inferior e superior quanto os registradores de base e limite devem ser protegidos. Eles não podem ser acessados em modo usuário. Obviamente, eles devem poder ser acessados em modo supervisor. O conteúdo desses registradores passa a fazer parte do contexto de execução dos processos. Podem ser mantidos no descritor de processo (DP) juntamente com as demais informações do seu contexto de execução. Quando ocorre um chaveamento de processo, os valores são copiados do DP para os registradores da MMU, limitando assim a região de memória a qual o processo que recebe o processador tem acesso.

Uma diferença básica entre as soluções das Figuras 6.2 e 6.3 está na carga dos programas. No esquema que emprega apenas registradores de limite, os programas são gerados para o endereço zero de memória. Entretanto, eles serão provavelmente carregados em um outro endereço da memória física. O endereço inicial do programa na memória física somente é conhecido no momento da carga, pois depende de quais outros programas estão sendo executados e de quanta memória eles ocupam. Dessa forma, no momento da carga, os endereços do programa devem ser corrigidos para que o programa execute corretamente no lugar onde foi colocado. Esse processo de correção de endereços é chamado de **relocação**. Um carregador que efetua uma relocação do programa em tempo de carga é chamado de **carregador relocador**.

No esquema que emprega registradores de base e limite, todos os programas são também gerados para o endereço zero de memória. Entretanto, eles podem ser carregados em qualquer lugar da memória física. Com o registrador de base contendo o endereço físico inicial, o programa funciona sem alterações em qualquer lugar da memória. Um carregador de programas que não precisa corrigir os endereços durante a carga é chamado de **carregador absoluto**. Nesse esquema, podemos considerar que ocorre uma relocação em tempo de execução, pois cada endereço sofre uma correção automática ao ser somado com o conteúdo do registrador de base.

A questão da relocação em tempo de carga ilustra a importância do suporte do hardware para o sistema operacional, nesse caso, o suporte que a MMU oferece para a gerência de memória. Ao longo deste capítulo, será visto que MMU mais sofisticadas permitem soluções mais eficientes para o problema da gerência da memória. O anexo A descreve as operações básicas realizadas por montadores, ligadores e carregadores.

6.2 Partições fixas

Partições fixas são a forma mais simples de gerência de memória para multiprogramação. A memória é primeiramente dividida em uma parte para uso do sistema operacional e uma parte para uso dos processos de usuários. A seguir, a parte dos usuários é dividida em várias partições de tamanhos diferentes porém fixos. Quando um programa deve ser carregado, é escolhida uma partição ainda livre. Obviamente, a partição deve ter um tamanho igual ou maior que o programa. Caso não exista uma partição livre que seja grande o bastante para conter o programa, ele não poderá ser executado no momento. Deverá esperar até que um dos processos em execução termine, liberando uma partição de tamanho suficiente. A Figura 6.4 mostra a memória com quatro partições de diferentes tamanhos, além da área reservada para o sistema operacional. Partições fixas podem ser implementadas tanto com registradores de limite superior e inferior quanto com registradores de base e limite.

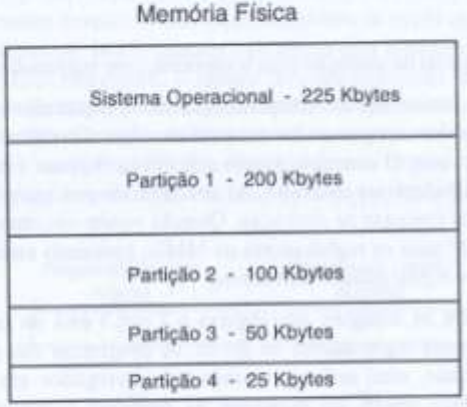

Figura 6.4 - Memória física dividida em partições fixas.

Existem dois problemas com esse tipo de gerência de memória. Dificilmente o programa a ser carregado terá o tamanho exato de uma partição. Ele será carregado em uma partição que é um pouco 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. O programa não pode ser executado devido à forma como a memória é gerenciada. Esse tipo de problema é chamado de **fragmentação externa**, isto é, memória perdida fora da área ocupada por um processo.

6.3 Partições variáveis

Quando **partições variáveis** são empregadas, o tamanho das partições é ajustado dinamicamente às necessidades exatas dos processos. Essa é uma técnica de gerência de memória mais flexível que partições fixas.

O sistema operacional mantém uma **lista de lacunas**, ou seja, de espaços livres na memória física. Quando um processo é criado, a lista de lacunas é percorrida. Será usada uma lacuna de tamanho maior ou igual ao tamanho do programa em questão. Entretanto, o que a lacuna original tiver a mais que o necessário para executar o programa será transformado em uma nova lacuna, apenas menor que a original. Dessa forma, o programa vai receber o tamanho exato de memória que necessita.

Existem quatro formas básicas de percorrer a lista de lacunas atrás de uma lacuna de tamanho suficiente. Os algoritmos ***first-fit*** e ***circular-fit*** são os mais utilizados. São eles:

- *First-fit*: utiliza a primeira lacuna que encontrar com tamanho suficiente;
- *Best-fit*: utiliza a lacuna que resultar na menor sobra;
- *Worst-fit*: utiliza a lacuna que resultar na maior sobra;
- *Circular-fit*: como *first-fit*, mas inicia a procura na lacuna seguinte à última sobra.

Quando um processo termina, a memória que ele ocupava é liberada. Isso corresponde à criação de uma nova lacuna. Caso a nova lacuna criada seja adjacente a outras lacunas, elas são unificadas. A Figura 6.5a mostra uma memória com três processos e uma lacuna. Existe um processo esperando para ser executado, mas não existe uma lacuna com tamanho suficiente. Suponha que o processo 2 termine. Sua área de memória é transformada em uma lacuna, o que permite que o programa do processo 4 seja carregado. A Figura 6.5b mostra a situação final.

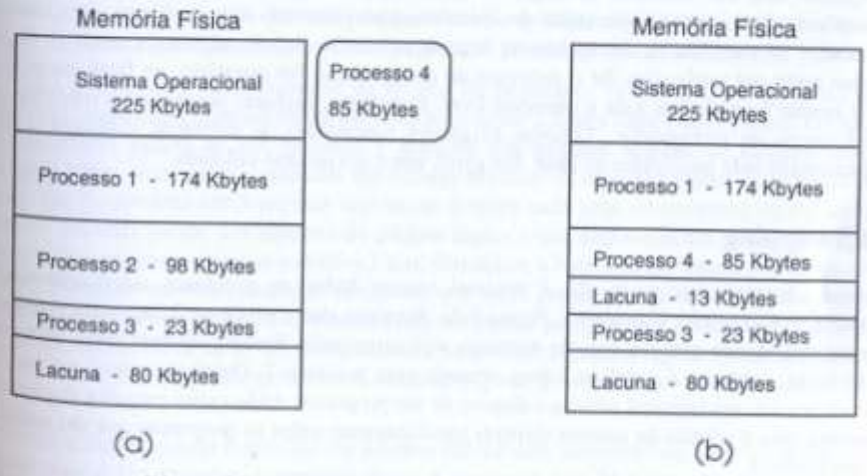

Figura 6.5 - Memória física dividida em partições variáveis.

Se o tamanho exato de cada programa é sempre alocado, não ocorre fragmentação interna. Entretanto, alocar sempre o tamanho exato pode gerar lacunas de alguns poucos bytes. Não é interessante arcar com o custo, em termos de memória e tempo de processamento, para manter lacunas de poucos bytes. Alguns sistemas organizam a memória em blocos de, por exemplo, 32 bytes. Esses blocos são muitas vezes chamados de **parágrafos**. A unidade de alocação passa a ser o parágrafo e o tamanho da área alocada por um processo deve ser um número inteiro de parágrafos, ou seja, um múltiplo exato de 32 bytes. Dessa forma, a menor lacuna possível terá o tamanho de um parágrafo ou 32 bytes. Nesse caso, poderemos ter uma fragmentação interna de até 31 bytes por processo.

Apesar da fragmentação interna introduzida, esse esquema possui várias vantagens. Em algumas arquiteturas, as variáveis do tipo inteiro devem ficar alinhadas corretamente na memória. Por exemplo, a arquitetura exige que inteiros de 4 bytes sejam posicionados necessariamente a partir de um byte com endereço par. A alocação de memória baseada em parágrafos facilita o atendimento das exigências de arquiteturas desse tipo. Também são necessários menos bits para endereçar uma lacuna na memória. Considerando uma memória com 1 Gbyte, são necessários 30 bits para endereçar um byte específico, mas apenas 25 bits para endereçar um parágrafo em particular.

Partições variáveis são tipicamente implementadas através de uma lista encadeada de lacunas. Cada lacuna é representada por um descritor de lacuna, que contém basicamente o seu endereço, tamanho e apontadores para as lacunas adjacentes. Como cada lacuna tem o tamanho mínimo de um parágrafo, ela pode hospedar o seu próprio descritor. Dessa forma, não é necessário alocar memória especialmente para uma "tabela de lacunas", pois a memória das próprias lacunas é usada para implementar a lista de lacunas. As lacunas são mantidas ordenadas pelo endereço, para facilitar a detecção de lacunas adjacentes, quando a situação acontecer.

Com partições variáveis a fragmentação externa é um problema grave. À medida que áreas de memória são alocadas e liberadas, muitos fragmentos são gerados. Ou seja, uma grande quantidade de lacunas pequenas demais para serem úteis. A memória adquire uma aparência de "queijo suíço". Não é incomum perder 1/3 da memória devido à fragmentação externa.

É possível tentar usar compactação de memória para eliminar esse problema. Processos são deslocados na memória de forma que as lacunas existentes fiquem adjacentes umas às outras e possam então ser unificadas. Se o processo de compactação for completo, ao final teremos uma única lacuna formada por toda a memória livre. Entretanto, deslocar processos na memória gasta muito tempo de processador. Também exige um mecanismo de relocação dinâmica, como o implementado pelo registrador de base. Em geral, não é um recurso utilizado.

6.4 Swapping

Existem situações nas quais não é possível manter todos os processos simultaneamente na memória. Por exemplo, considere a Figura 6.5a. Suponha que o processo 2 faça uma chamada de sistema, solicitando que sua área de memória seja aumentada. Embora existam áreas de memória ainda livres, nenhuma é contígua à área ocupada pelo processo 2. Outro exemplo é a situação na qual um usuário em terminal solicita o disparo de um programa, e não existe memória disponível no momento, mas é política do sistema disparar imediatamente todos os programas que são solicitados via um terminal.

Uma solução para essas situações é o mecanismo chamado de ***swapping***. A gerência de memória reserva uma área do disco para o seu uso. Em determinadas situações, um processo é completamente copiado da memória para o disco. Sua execução é suspensa, ou seja, seu descritor

de processo é removido da fila do processador e colocado em uma fila de processos suspensos. É dito que esse processo sofreu um ***swap-out***. Mais tarde, ele sofrerá um ***swap-in***, ou seja, será copiado novamente para a memória. Seu descritor de processo volta então para a fila do processador, e sua execução será retomada. O resultado desse revezamento no disco é que o sistema operacional consegue executar mais processos do que caberia em um mesmo instante na memória.

Swapping impõe aos programas um grande custo em termos de tempo de execução. Copiar todo o processo da memória para o disco e mais tarde de volta para a memória é uma operação demorada. É necessário deixar o processo um tempo razoável no disco para justificar tal operação. Por exemplo, em torno de alguns segundos.

Em sistemas nos quais uma pessoa interage com o programa durante a sua execução (chamado antigamente de modo *timesharing*), o mecanismo de *swapping* somente é utilizado em último caso, quando não é possível manter todos os processos na memória. A queda no desempenho do sistema é imediatamente sentida pelo usuário no terminal. Para processos que são executados em *background*, ou seja, desvinculados de um terminal, o mecanismo torna-se mais aceitável.

Swapping pode ser usado tanto com partições fixas quanto com partições variáveis. Caso o processo, no momento do *swap-in*, volte para uma posição diferente de memória, é necessário corrigir os seus endereços. Isso não é necessário quando se usa um mecanismo baseado em registrador de base, como mostrado na Figura 6.3. Nesse caso, basta corrigir o conteúdo do registrador de base, e o programa executará corretamente em sua nova posição na memória física.

6.5 Paginação

A técnica de partições fixas gera muita perda de memória e não é mais utilizada na prática. Embora partições variáveis seja um mecanismo mais flexível, 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 restrição for eliminada, ou seja, permitir que um programa ocupasse áreas não contíguas de memória, não haveria fragmentação externa. A técnica de **paginação** possibilita exatamente isso.

A Figura 6.6 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. Considere o byte Y2. Ele possui o endereço lógico 00101. Podemos ver esse endereço composto por duas partes. Os primeiros 3 bits indicam o número da página, isto é, 001. Os últimos 2 bits indicam a posição de Y2 dentro da página, isto é, 01. Observe que todos os bytes pertencentes a uma mesma página lógica apresentam o mesmo número de página. De forma semelhante, todas as páginas possuem bytes com deslocamento entre 00 e 11.

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 um deslocamento dentro dessa página física.

Um programa é carregado 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. Mais do que isso, a ordem em que as páginas lógicas aparecem na memória física pode ser qualquer, não precisa ser a mesma da memória lógica. Observe que, no exemplo, o conteúdo das páginas lógicas foi espalhado pela memória física, mantendo o critério de que cada página lógica é carregada em exatamente uma página física. A página lógica 000 (X1, X2, X3 e X4) foi carregada na página física 010. A página lógica 001 (Y1, Y2, Y3 e Y4) foi carregada na página física 101. A página lógica 010 (Z1, Z2, Z3 e Z4) foi carregada na página física 000.

Durante a carga é montada uma **tabela de páginas** para o processo. Essa tabela informa, para cada página lógica, qual a página física correspondente. No exemplo, a tabela é formada por 3 entradas, uma vez que o processo possui 3 páginas lógicas.

Quando um processo executa, ele manipula endereços lógicos. O programa é escrito com a suposição que ele vai ocupar uma área contígua de memória, que inicia no endereço zero, ou seja, vai ocupar a memória lógica do processo. Para que o programa execute corretamente, é necessário transformar o endereço lógico especificado em cada instrução executada, no endereço físico correspondente. Isso é feito com o auxílio da tabela de páginas.

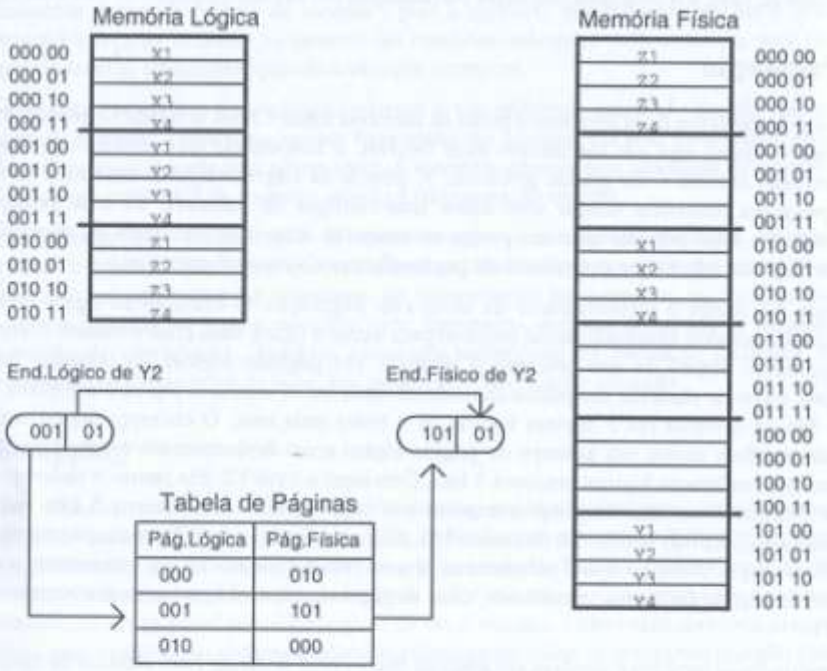

Figura 6.6 - Mecanismo básico de paginação.

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. Dessa forma, é obtido o número da página física correspondente. 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 6.6 mostra como o endereço lógico do byte Y2 é transformado no endereço físico correspondente. O endereço lógico 00101 é dividido em número de página lógica 001 e deslocamento 01. A entrada 001 da tabela de páginas indica que essa página lógica foi carregada na página física 101. Finalmente, as duas partes são unidas, formando o endereço físico 10101.

Na prática, os tamanhos de página variam entre 1 Kbytes e 8 Kbytes. Espaços de endereçamento lógico variam de 64 Kbytes para sistemas antigos até muitos Gbytes para máquinas atuais. Espaços de endereçamento físico também ficam, em geral, na ordem de Gbytes. Note que o espaço de endereçamento físico denota a capacidade de endereçamento do processador, e não a quantidade de memória realmente instalada na máquina. Embora o processador de um computador doméstico atual possa endereçar 32 Gbytes, tipicamente a memória física realmente instalada é menor que 1 Gbyte.

Na paginação, uma página lógica pode ser carregada em qualquer página física que esteja livre. Dessa forma, não existe fragmentação externa. Como a unidade de alocação é a página, e que o tamanho de um processo raramente é igual a um múltiplo do tamanho de página, isto introduz uma fragmentação interna. Suponha que um sistema no qual as páginas são de 4 Kbytes, e um programa necessita 201 Kbytes para executar. Serão alocadas para ele 51 páginas, totalizando 204 Kbytes. Isso resultará em uma fragmentação interna de 3 Kbytes. Em média, podemos esperar uma fragmentação interna de meia página por processo.

Existem vantagens e desvantagens em utilizar páginas grandes. Páginas maiores significam que um processo terá menos páginas, a tabela de páginas será menor, a leitura do disco será mais eficiente. Em geral, páginas maiores resultam em um custo menor imposto pelo mecanismo de gerência de memória, ou seja, um ***overhead*** menor. Por outro lado, páginas maiores resultam em uma fragmentação interna maior. Normalmente não é o sistema operacional que escolhe o tamanho das páginas. Esse valor é fixado pelo hardware que suporta a gerência de memória, ou seja, pela MMU do computador em questão.

A gerência de memória deve manter controle das áreas ainda livres na memória. Isso significa manter uma lista de páginas físicas livres. Uma forma de implementar esse controle é através de um mapa de bits. Cada bit representa o estado de uma página física em particular. Por exemplo, 0 indica página livre e 1 indica página ocupada. Para localizar uma página física livre, basta percorrer o mapa de bits até encontrar um bit 0. A posição do bit no mapa indica o número da página física livre. Esse mecanismo poderá ficar muito lento quando a memória física for grande e estiver praticamente toda ocupada. Uma alternativa é manter uma lista encadeada com os números das páginas físicas livres. Para localizar uma página física livre basta pegar o primeiro elemento dessa lista. As próprias páginas livres da memória podem ser utilizadas para armazenar a lista.

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. Os próximos parágrafos discutem três formas usadas para implementar a tabela de páginas, sendo a última delas a mais usada atualmente.

Quando a tabela de páginas é pequena, ela pode ser completamente colocada em registradores de acesso rápido. Por exemplo, um computador no qual a memória lógica dos processos seja de apenas 64 Kbytes e cada página ocupe 8 Kbytes, terá apenas 8 entradas nas tabelas de páginas. Registradores apresentam a vantagem de serem rápidos e, portanto, não degradam o tempo de acesso à memória. Quando ocorre um chaveamento de processos, a tabela de páginas do processo que recebe o processador deve ser copiada do descritor de processo (DP) para os registradores.

Quando a tabela de páginas é muito grande, não é possível mantê-la em registradores. Uma outra solução é manter a tabela de páginas na própria memória. A MMU possui então dois registradores para localizar a tabela na memória. O **registrador de base da tabela de páginas** (*Page Table Base Register*, PTBR) indica o endereço físico de memória onde a tabela está colocada. O **registrador de limite da tabela de páginas** (*Page Table Limit Register*, PTLR) indica o número de entradas da tabela. O problema desse mecanismo é que agora cada acesso que um processo faz à memória lógica transforma-se em dois acessos à memória física. No primeiro acesso, a tabela de páginas é consultada, e o endereço lógico é transformado em endereço físico. No segundo acesso, a memória do processo é lida ou escrita. Quando ocorre um chaveamento de processos, os valores do PTBR e do PTLR para a tabela de páginas do processo que recebe o processador devem ser copiados do DP para os registradores na MMU.

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**). O acesso a essa *cache* é rápido e não degrada o tempo de acesso à memória como um todo. Quando a entrada requerida da tabela de páginas está na TLB, o acesso à memória lógica do processo é feito com um único acesso à memória física, como quando registradores são utilizados. Nesse caso, é dito que tivemos um acerto (*hit*). Quando a entrada da tabela de páginas associada com a página lógica acessada não está na TLB (*miss*), é necessário um duplo acesso à memória física, como no esquema anterior. Nesse caso, a entrada referente a página lógica acessada é incluída na TLB, na suposição (correta na maioria das vezes) de que o processo acessará essa página mais vezes logo em seguida. Os próximos acessos já encontrarão essa entrada da tabela de páginas na TLB, e o acesso será rápido.

Normalmente, a memória *cache* é implementada através de um componente de hardware conhecido como memória associativa. A memória associativa inclui não somente algumas células de memória, mas também toda a eletrônica necessária para fazer uma pesquisa paralela, incluindo todas as células. O resultado é um hardware caro, o que limita o tamanho das memórias associativas utilizadas na prática. Entretanto, mesmo memórias associativas de 16 a 32 entradas permitem taxas de acerto de 80% a 90%. O resultado final é um tempo de acesso à memória com paginação que é apenas 20% a 30% maior do que seria caso não houvesse paginação.

Quando memória *cache* é usada e ocorre um chaveamento de processos, novamente os valores do PTBR e do PTLR para a tabela de páginas do processo que recebe o processador devem ser copiados do DP para os registradores na MMU. Além disso, a memória *cache* deve ser esvaziada (*flushed*). Isso é necessário, pois ela ainda contém entradas da tabela de página do processo que executou antes e agora perdeu o processador. Esquemas alternativos incluem o número do processo em cada entrada da TLB para resolver esse problema.

Normalmente, cada entrada da tabela de páginas possui, além do número da página física correspondente, uma série de bits que auxiliam na gerência da memória. É comum a inclusão de um bit de válido/inválido. Quando o processo tenta acessar uma página que está marcada como inválida, a MMU gera uma interrupção de proteção, e o sistema operacional é acionado. Em geral, isso representa um erro de programação, pois o processo está tentando acessar uma página que não faz parte da sua memória lógica. Entretanto, existem também outros usos para esse bit, como será visto no capítulo sobre memória virtual.

Também é comum a inclusão de bits que indicam como o conteúdo daquela página pode ser usado. Tipicamente, uma página pode ser para apenas leitura (*read-only*, RO), para apenas execução (*execute-only*, XO) ou para leitura e escrita (*read-write*, RW). Caso o processo tente acessar a página de uma maneira diferente daquela determinada pelos bits de proteção, a MMU gera uma interrupção de proteção e aciona o sistema operacional. Esse mecanismo permite, entre outras coisas, que erros de programação sejam detectados.

É importante observar que a proteção entre processos é facilmente conseguida com uma MMU que suporte paginação. Em primeiro lugar, o mecanismo de paginação garante que cada processo somente tenha acesso às páginas físicas que constam em sua tabela de páginas. Essa tabela de páginas é construída pelo sistema operacional e fica em uma região da memória à qual apenas o sistema operacional tem acesso. O acesso aos registradores PTBR e PTLR é privilegiado, isto é, restrito ao código do sistema operacional, que executa em modo supervisor. Finalmente, o sistema operacional ainda dispõe dos bits de proteção na tabela de páginas para controlar como cada página é acessada e para marcar como inválidas as entradas da tabela de páginas que o processo não pode usar.

Em sistemas atuais, a tabela de páginas pode ser muito grande. Considere como exemplo o processador 80386 da Intel que trabalha com endereços de 32 bits. O espaço de endereçamento lógico pode ser de até 4 Gbytes (1 Gbyte equivale a 1024×1024×1024 bytes, ou ainda, 2 elevado a 30 bits). Cada página ocupa 4 Kbytes. Isso significa que um processo pode ter até 1.048.576 entradas na tabela de páginas, o que representaria uma tabela de páginas ocupando 4 Mbytes de memória (cada entrada são 4 bytes).

Em arquiteturas atuais, uma tabela de páginas completa raramente é utilizada. 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.

No caso do Intel 386, a tabela de páginas é dividida em dois níveis. No primeiro nível, existe um diretório de tabelas de páginas. No segundo nível, estão os pedaços, chamados de tabelas de páginas na documentação da Intel. O endereço lógico é formado por 32 bits, dividido em número da entrada no diretório de tabelas de páginas (10 bits), número da página na tabela de páginas indicada (10 bits) e deslocamento dentro da página (12 bits). A Figura 6.7 ilustra como um endereço lógico é transformado em endereço físico.

Uma alternativa seria ter toda a tabela de páginas ocupando uma área contígua de memória, dispensando o diretório, mas isso significaria usar alocação contígua de memória, o que traria problemas para o crescimento dinâmico da memória dos processos. Uma outra vantagem do esquema em dois níveis é a existência de regiões não contínuas de espaço lógico. Por exemplo, de 00000000H até 00010000H fica o programa, e de FFFF0000H até FFFFFFFH fica a pilha desse

programa. Nesse caso, uma tabela de páginas completa e contígua teria um enorme espaço inútil entre os dois extremos ocupados.

Em resumo, a implementação da tabela de páginas em dois níveis traz as seguintes vantagens:

- A tabela cresce de pedaço em pedaço, à medida que a alocação de páginas físicas acontece;
- Apenas o necessário é usado, com uma pequena **fragmentação interna** na própria tabela, quando não são necessárias todas as entradas do pedaço alocado;
- O processo pode "povoar" diferentes regiões do espaço lógico, sem dificuldades.

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 [JAC98a].

6.6 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. Outros sistemas trabalham com uma granularidade menor. Por exemplo, cada objeto ou módulo corresponde a um segmento. Em geral, o programador atribui nomes aos segmentos, e o compilador transforma esses nomes em números. Por exemplo, o segmento "Sub-rotinas da biblioteca gráfica" passa a ser conhecido como segmento número 5.

É possível orientar a gerência de memória para suportar diretamente o conceito de segmento. Nesse caso, a memória lógica do processo passa a ser organizada em termos de segmentos. 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. A Figura 6.8 ilustra essa técnica de gerência de memória. Todos os números mostrados são valores binários.

Os processos geram endereços lógicos compostos por um número de segmento e um deslocamento dentro do segmento. A MMU inicialmente utiliza o número de segmento fornecido para indexar a tabela de segmentos. Caso esse segmento não exista, é gerada uma interrupção de proteção. Uma vez localizada a entrada na tabela de segmentos, o deslocamento fornecido é comparado com o limite do segmento. Um deslocamento maior que o limite significa que o processo está tentando endereçar além do final do segmento, fora da sua memória lógica, e uma interrupção de proteção é gerada. Finalmente, o deslocamento fornecido é somado ao valor de base do segmento, resultando no endereço físico correspondente ao endereço lógico fornecido.

Considerando a Figura 6.8, suponha que o processo gere o endereço lógico do byte D3, ou seja, segmento 01 e deslocamento 00010. A tabela de segmentos informa que o segmento 01 possui base 00000 e limite 0100. O deslocamento de 00010 é válido, pois é menor que o limite 0100. Somando a base do segmento 00000 com o deslocamento 00010, temos o endereço físico 00010. Esse é o endereço do byte D3 na memória física.

A tabela de segmentos pode ser implementada através das mesmas três formas básicas que foram apresentadas na seção sobre paginação. Os bits de proteção existentes nas tabelas de páginas também são usados em tabelas de segmentos. 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.

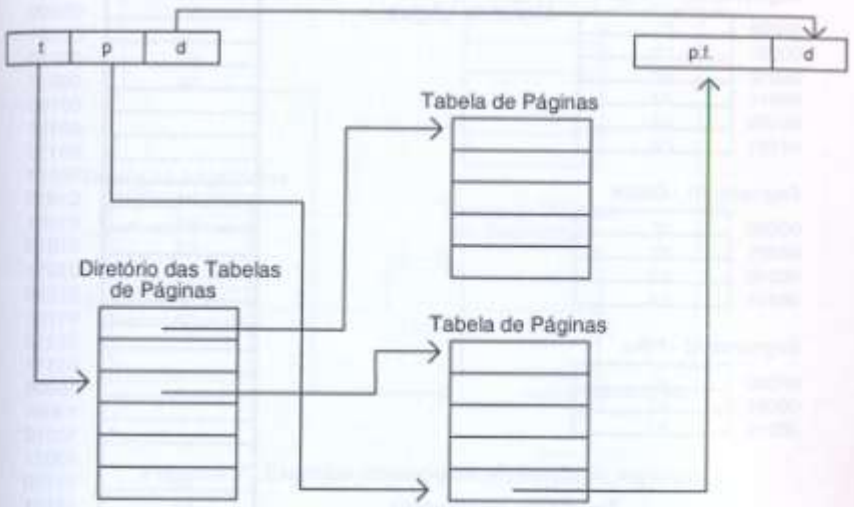

Figura 6.7 - Tabela de páginas organizada em dois níveis.

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. Podemos citar como exemplos o código de uma sub-rotina e uma pilha, respectivamente.

Por exemplo, suponha que o código das rotinas de biblioteca de uma linguagem de programação é compilado como sendo um segmento único. Esse segmento é marcado como "para apenas execução", ou seja, não pode ser lido nem escrito. Todos os programas escritos nessa linguagem de programação utilizam esse segmento. Entretanto, apenas uma cópia dele é necessária na memória física. Todos os processos executando programas escritos nessa linguagem terão em sua respectiva tabela de segmentos uma referência à posição desse segmento na memória física. Como ele nunca é alterado, uma única cópia na memória física é suficiente para atender a todos os processos. Nesse caso, o sistema operacional deve manter uma tabela na memória, indicando quais segmentos estão na memória principal e qual a sua localização. Um segmento compartilhado somente é removido da memória principal quando nenhum processo o estiver usando, isto é, ele não consta mais em nenhuma tabela de segmentos em uso.

Se considerarmos um sistema com dezenas de programas executando, e a maioria deles compartilhando as mesmas bibliotecas, poderemos perceber a importância do compartilhamento de código. Também é necessário salientar a importância dos bits de proteção no momento de restringir o acesso dos processos aos segmentos compartilhados. Obviamente, um segmento contendo dados de um programa é do tipo "para leitura e escrita" e não pode ser compartilhado.

Tabela de Segmentos

Segmento	Base	Limite
00	01000	0110
01	00000	0100
10	10100	0011

Figura 6.8 - Gerência de memória baseada em segmentos.

6.7 Segmentação paginada

Com a segmentação, volta o problema da fragmentação externa. A sucessiva alocação e liberação de segmentos com diferentes tamanhos gera lacunas pequenas demais para serem úteis. 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. Cada segmento possui uma tabela de páginas associada. No momento de endereçar a memória, a tabela de segmentos indica, para cada segmento, onde a respectiva tabela de páginas está. Essa tabela de páginas é usada para transformar o endereço de página lógica de determinado segmento em endereço de página física, como é feito normalmente na paginação. A Figura 6.9 ilustra essa configuração básica. É importante salientar que essa não é a única maneira de construir uma solução para a segmentação paginada.

A alocação de espaço em memória é feita na base de página física, eliminando completamente o problema de fragmentação externa. Entretanto, o sistema passa a ter fragmentação interna. Em média, teremos meia página de fragmentação interna por segmento de processo.

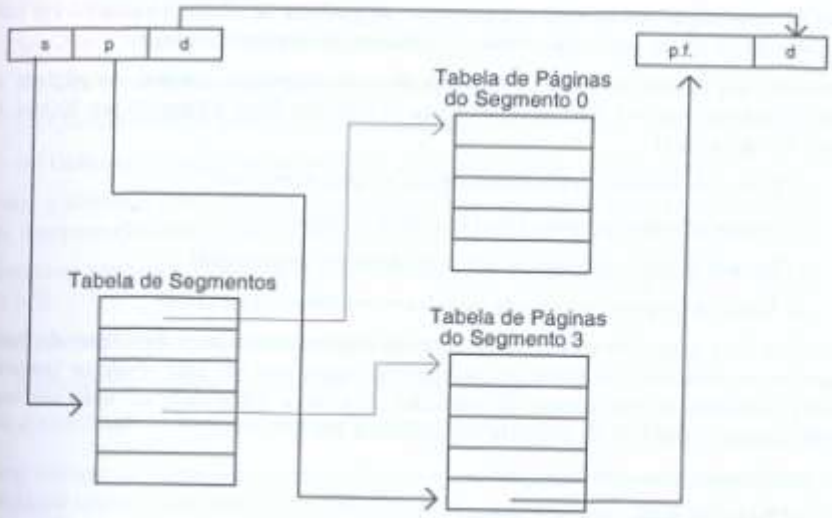

Figura 6.9 - Esquema clássico para segmentação paginada.

6.8 Exercícios

1) Considere um sistema cuja gerência de memória é feita através de partições variáveis. Nesse momento, existem as seguintes lacunas (áreas livres): 10K, 4K, 20K, 18K, 7K, 9K, 12K e 13K, nessa ordem. Quais espaços serão ocupados pelas solicitações: 5K, 10K e 6K, nessa ordem, se: (Seção 6.3)

(a) *First-Fit* for utilizado?

(b) *Best-Fit* for utilizado?

(c) *Worst-Fit* for utilizado?

(d) *Circular-Fit* for utilizado?

2) Considere novamente um sistema cuja gerência de memória utiliza partições variáveis. Nesse momento, existem as seguintes lacunas (áreas livres): 10K, 4K, 20K, 18K, 7K, 9K, 12K e 13K, nessa ordem. Quais espaços serão ocupados pelas solicitações: 15K, 4K e 8K, nessa ordem, se: (Seção 6.3)

(a) *First-Fit* for utilizado?

(b) *Best-Fit* for utilizado?

(c) *Worst-Fit* for utilizado?

(d) *Circular-Fit* for utilizado?

3) Compare partições fixas, partições variáveis, paginação simples e segmentação simples com respeito ao compartilhamento de memória (código). Para cada uma, mostre como é possível implementar ou por que é impossível implementar. Lembre-se de que é sempre necessário manter a proteção entre usuários, e que os programas ocupam um espaço lógico contíguo.

4) Qual a fragmentação apresentada pelos métodos de gerência de memória baseados em partições fixas, partições variáveis, paginação simples e segmentação simples? Justifique.

5) Considere um sistema operacional que trabalha com paginação simples. As páginas são de 1Kbyte. O endereço lógico é formado por 16 bits. O endereço físico é formado por 20 bits. Qual o tamanho do: (Seção 6.5)

(a) Espaço de endereçamento lógico (maior programa possível)?

(b) Espaço de endereçamento físico (memória principal)?

(c) Entrada da tabela de páginas, sem considerar bits de proteção?

(d) Tabela de páginas (número de entradas necessárias no pior caso)?

6) Partições fixas e partições variáveis podem ser implementadas com dois tipos de hardware: baseado em registradores de limite ou baseado em registrador de base. Pode-se também usar swapping associado ao mecanismo de partições. Discuta a viabilidade de usar um ou outro hardware, quando a gerência de memória for: (Seções 6.2/6.3)

(a) Partições fixas com swapping;

(b) Partições variáveis com swapping.

7) O professor de sistemas operacionais passou como trabalho para os seus alunos a construção de um ligador para um novo sistema que será desenvolvido no próximo semestre. O computador para o qual será construído o novo sistema operacional foi encomendado junto ao IPIAL (Instituto de Pesquisas em Informática de Algum Lugar), mas ainda não chegou. O hardware para acesso à memória ainda não é conhecido. Ele deverá ser um dos seguintes:

H1 - Apenas dois registradores de limite para proteção da memória;

H2 - Um registrador de base mais um de limite;

H3 - Hardware suficiente para implementar paginação.

Em função disso, a gerência de memória a ser utilizada no novo sistema ainda não está definida. Ela poderá ser baseada em partições fixas, partições variáveis ou paginação.

Isso significa que ainda não está claro que tipo de arquivo executável deverá ser gerado para o novo sistema. Existem 3 possibilidades:

X1 - Código gerado para o endereço 0, acompanhado de mapa para relocação;

X2 - Código gerado para o endereço 0, sem mapa de relocação;

X3 - Código gerado para o endereço de carga correto.

Analise a viabilidade de cada tipo de arquivo executável para cada uma das gerências de memória listadas abaixo (são 27 combinações ao todo):

(a) Partições fixas com registradores de limite;

(b) Partições variáveis com registrador de base e limite;

(c) Paginação com hardware para paginação, obviamente.

8) O sistema operacional XYZ utiliza paginação como mecanismo de gerência de memória. São utilizadas páginas de 1Kbyte. Um endereço lógico ocupa 20 bits. Um endereço físico ocupa 24 bits. Cada entrada na tabela de páginas contém, além do número da página física, um bit de válido/inválido e um bit que indica apenas leitura (*read-only*). Mostre como podem ser calculados os seguintes valores: (Seção 6.5)

(a) Qual o tamanho máximo para a memória física.

(b) Qual o maior programa que o sistema suporta.

(c) Quantas entradas possui a tabela de páginas.

(d) Quantos bits serão necessários para a tabela de páginas (cálculo exato).

9) Mostre a diferença entre fragmentação interna e fragmentação externa. Em que situação poderá ocorrer fragmentação interna quando partições variáveis são utilizadas? (Seção 6.3)

10) Considere um sistema que utiliza partições variáveis na gerência da memória. Para os itens: (Seção 6.3)

(a) Arquitetura emprega apenas registradores de limite;

(b) Arquitetura utiliza registradores de base e de limite;

Analise a viabilidade de ser implementado *swapping*. Defina o tipo de carregador a ser utilizado.

11) Uma vez que paginação não apresenta fragmentação externa enquanto segmentação apresenta fragmentação externa, qual motivação existe para justificar a idéia de segmentação ? Que vantagem a segmentação pura apresenta sobre a paginação pura ? (Seções 6.5/6.6)

12) Em um sistema usando segmentação paginada, o espaço de endereçamento lógico de cada processo consiste de no máximo 16 segmentos, cada um deles podendo ter até 64 Kbytes de tamanho. As páginas físicas são de 512 bytes. Diga quantos bits são necessários para especificar cada uma das grandezas abaixo, explicando de onde veio cada número. (Seção 6.7)

(a) Número do segmento;

(b) Número de uma página lógica dentro do segmento;

(c) Deslocamento dentro de uma página;

(d) Endereço lógico completo.

áreas vão sendo alocadas. O descritor de arquivo possui uma tabela com N entradas. Cada entrada é composta pelo endereço de uma área alocada e pelo tamanho dessa área. (Seção 8.3)

(a) Qual o espaço de disco gasto para localização do arquivo?

(b) Como pode ser implementado o acesso relativo?

5) Um disco CDROM contém um sistema de arquivos no qual todos os arquivos são imutáveis. Qual método de alocação, entre alocação contígua, encadeada e indexada, é o mais apropriado? Justifique sua resposta. (Seção 8.3)

6) Explique o mecanismo de dois níveis de tabelas para manter as informações sobre arquivos abertos no sistema. Por que não é usada apenas uma tabela geral? (Seção 8.3)

7) Como é possível evitar que um processo abra um arquivo apenas para leitura e depois execute operações de escrita? Qual a estrutura de dados envolvida? (Seção 8.3

9
Linux

No decorrer deste livro foram apresentados diversos conceitos fundamentais de sistemas operacionais e muitas vezes diferentes métodos para solucionar um determinado problema, mas, e na vida "real"? Como as coisas são feitas? Este capítulo tem por objetivo justamente responder essa pergunta através do estudo de um sistema operacional que está tornando cada vez mais um fenômeno de popularidade: o GNU/Linux.

O sistema GNU/Linux é um sistema livre que pode ser facilmente obtido na Internet. O Linux é sistema UNIX-*like* e emprega muitas soluções comuns a outros sistemas tipo UNIX. Neste capítulo, nós apresentaremos um breve histórico do Linux para, em seguida, estudarmos detalhes de sua arquitetura interna, ilustrando como o Linux implementa os conceitos estudados neste livro. Nós iniciaremos esse estudo com o conceito de processos. Em seguida, veremos como o Linux gerencia o recurso processador, analisando o seu mecanismo de escalonamento. A etapa seguinte será a gerência de memória e a memória virtual, passando pelo seu mecanismo de *swap*. Na seqüência, o estudo do sistema de arquivos do Linux, para finalmente concluir com alguns aspectos relacionados à gerência de entrada e saída.

9.1 Introdução: um pouco de história, distribuições e versões

A origem do Linux é bastante modesta. Ele foi desenvolvido pela iniciativa de um estudante finlandês chamado Linus Torvalds. A principal motivação de Linus foi a sua decepção pessoal com o sistema `minix`. O `minix` é um sistema simples, baseado em UNIX, desenvolvido para fins didáticos e amplamentente difundido no meio acadêmico no início dos anos 90. Linus teve então a idéia de escrever, a partir do zero, um novo núcleo de sistema operacional, também baseado na filosofia UNIX, para suprir os problemas que ele via no `minix`.

Após a conclusão de uma versão inicial de seu novo sistema operacional, Linus divulgou-o através da lista de discussão do próprio `minix`, disponibilizando-o para que outros estudantes utilizassem-no como fonte de estudo. Rapidamente, outras pessoas aderiram a esse projeto, oferecendo sua contribuição pessoal para o desenvolvimento de novas funcionalidades, depuração, etc. Essa iniciativa tornou-se o fenômeno Linux que todos conhecemos atualmente. Hoje em dia, milhares de programadores auxiliam a desenvolver não somente o núcleo do Linux, mas também ferramentas que facilitam sua utilização e divulgação, assim como os mais variados programas de sistemas. Linus Torvald ainda participa da evolução do Linux como uma espécie de gerente geral de projetos da parte relacionada ao núcleo. Assim surgiu o Linux (Linus' UNIX).

9.1.1 As distribuições Linux

O Linux é um clone do UNIX derivado em grande parte da família BSD. Tecnicamente, o nome Linux faz referência ao núcleo do sistema operacional, mas esse nome atualmente designa também um conjunto completo de software, também software livre, oriundo das mais diferentes partes do mundo. Esse conjunto de software, mais o núcleo Linux, constituem o que se chama de **distribuição**. Existem várias distribuições. Cita-se por exemplo, RedHat, Debian, Slackware, SuSe, Mandrake, Conectiva (essa última uma distribuição brasileira).

Todas as distribuições oferecem uma versão do núcleo Linux, um conjunto de software e uma interface de instalação. A diferença entre elas reside justamente nesses dois últimos pontos. O conjunto de software varia de distribuição para distribuição. Elas propõem ainda diferentes métodos de instalação e manutenção do sistema. A escolha de uma distribuição depende basicamente do gosto pessoal e da "intimidade" que o usuário tem com os comandos do sistema UNIX. Durante muito tempo, as distribuições Debian e SuSe eram tidas como as distribuições para "iniciados" em UNIX, ao passo que a distribuição RedHat era associada a usuários mais leigos.

Apesar do conjunto de software ser diferente de uma distribuição para outra, existe uma base comum bastante grande. Essa base corresponde às ferramentas desenvolvidas dentro do projeto GNU (GNU *is Not Unix*) da Free Software Foundation. Entre essas, pode-se citar o compilador C (gcc), o editor de textos emacs, o editor gráfico gimp, o ambiente de janelas gnome, etc. A lista de softwares GNU é imensa.

Outro ponto em relação às distribuições GNU/Linux é a questão comercial. Se o GNU/Linux é um sistema livre, por que nós compramos uma distribuição? Na realidade, o que se paga, no momento da compra de uma distribuição, não é o GNU/Linux em si, mas sim uma série de serviços que são disponibilizados pela distribuição. Você está comprando o conforto de ter um CDROM com a instalação, de possuir um manual de utilização e/ou instalação, eventualmente um suporte do tipo *hot-line*, ou ainda algum software específico desenvolvido e vendido junto com a distribuição. Normalmente, é de praxe disponibilizar em um site na Internet as imagens dos CDROMs de instalação para serem copiadas, ou ainda, permitir a instalação do GNU/Linux via Internet (ftp).

9.1.2 As versões do núcleo Linux

O Linux é um núcleo de sistema operacional em constante evolução. No mundo, neste instante, milhares de programadores, de curiosos, de *hackers* estão usando, analisando, descobrindo e relatando eventuais *bugs*. Isto faz com o núcleo Linux tenha uma qualidade bastante grande. Além disto, através de grupos de trabalho, pessoas trabalham voluntariamente para inserir novas funcionalidades no seu núcleo, propondo modificações em estruturas de dados e em algoritmos internos para fornecer ao núcleo Linux um melhor desempenho.

Considerando esse aspecto evolutivo do núcleo Linux, torna-se imprescindível manter um controle da evolução do núcleo. Isto é feito através da distribuição de dois tipos de núcleo: um dito núcleo em desenvolvimento, outro dito núcleo estável. Um **núcleo em desenvolvimento** é justamente aquele em que os programadores têm liberdade para alterar substancialmente seu código através da inserção de novas técnicas de programação, da alteração dos principais algoritmos, etc., ou seja, é um núcleo de experimentação. Os usuários que optam por utilizar um núcleo de desenvolvimento correm o risco de se defrontar com comportamentos estranhos e eventualmente até errados. Após um tempo de maturação e de depuração as versões em desenvolvimento dão origem as versões estáveis.

Um **núcleo estável** é aquele que é considerado pela comunidade de desenvolvedores como "confiável" e então liberado para uso para toda comunidade Internet. O fato a expressão confiável estar entre aspas refere-se a que um sistema operacional, sendo um software de uma complexidade e tamanho bastante grande, sempre existem alguns *bugs* menores a serem corrigidos. Nesse ponto, entra o que se denomina de *releases*. Uma ***release*** aporta correções a esses *bugs* menores em um núcleo. As *releases* estão associadas tanto às versões em desenvolvimento quanto às estáveis.

O núcleo Linux diferencia suas versões estáveis de suas versões de desenvolvimento de acordo com um simples sistema de numeração. Cada versão é caracterizada por 3 números separados por um ponto, como por exemplo, 2.2.14. Os primeiros dois números são empregados para identificar a versão propriamente dita. O terceiro número identifica a *release*. O primeiro número indica uma troca radical no núcleo e de sua filosofia de projeto. Até hoje existem três números de série: 0, 1 e 2.

As versões da série 0 correspondem ao nascimento do núcleo Linux. Apesar de possuir funcionalidades comuns de sistemas operacionais, essa versão continha muitas limitações em relação de recursos, como por exemplo, o fato de o único sistema de arquivos suportado ser o do `minix`. As versões da série 1 são um marco importante na história do Linux. Foi a partir da versão 1 que o Linux incorporou funcionalidades de rede. Essa série evoluiu com a inclusão do suporte a vários sistemas de arquivos, *drivers* para diferentes dispositivos, memória virtual, entre outras melhorias. As versões da série 2 correspondem aos núcleos atuais. Aqui, os fatos marcantes são a inclusão de suportes a processadores não-INTEL (Sparc, Alpha, etc.) e o início da preocupação em desenvolver um núcleo preparado para máquinas multiprocessadoras (SMPs) através do emprego de *multithreading*.

O segundo número de um código de versão é empregado para identificar se a versão é de desenvolvimento ou estável. Se esse número for ímpar, a versão corresponde a um código em desenvolvimento; se par, a uma versão estável. O terceiro número indica a *release*. No momento em que este livro[6] estava sendo revisado, a versão estável comum em distribuições GNU/Linux era a 2.6.5.

9.2 Arquitetura de sistemas operacionais

Tradicionalmente, os núcleos de sistemas operacionais são organizados de duas formas diferentes: monolítico e micronúcleo. A maioria dos sistemas operacionais UNIX tem seu núcleo organizado segundo uma estrutura dita monolítica. Um **núcleo monolítico** é aquele em que todos os componentes do sistema operacional fazem parte de um código único. Esse código único pode ser visto como o "processo sistema operacional" que executa em modo protegido. Um sistema operacional que segue a filosofia **micronúcleo** (*microkernel*) possui um conjunto reduzido de funcionalidades tais como primitivas de sincronização, um escalonador simples e um mecanismo de comunicação entre processos. Os demais componentes necessários ao sistema operacional como gerência de memória, sistemas de arquivos, *drivers* de dispositivos, etc., são implementados através de processos separados que interagem entre si e com o micronúcleo através de troca de mensagens.

Hoje em dia, embora o Windows NT possa ser visto como um contra-exemplo, os sistemas operacionais baseados em micronúcleo são restritos à pesquisa acadêmica. Normalmente um sistema baseado em micronúcleo apresenta um desempenho menor se comparado aos núcleos monolíticos devido justamente à necessidade de comunicação entre os diferentes componentes

[6] março de 2004.

(subsistemas) e o micronúcleo. A grande vantagem do micronúcleo é sua estruturação em componentes e o emprego de conceitos relacionados com programação orientada a objetos. Esta característica oferece, a baixo custo, um alto grau de portabilidade. Todas as particularidades do hardware estão "escondidas" no micronúcleo. Outra vantagem de sistemas operacionais baseados em micronúcleos é o uso otimizado de recursos de memória. Os processos que correspondem a subsistemas, a partir do momento em que não são utilizados, não são carregados em memória; ou melhor ainda, se deixam de ser utilizados, sofrem *swap* ou são destruídos, liberando assim recursos para processos de usuários.

Na realidade, existe um meio termo que é utilizado no Linux: os módulos. Um **módulo** é um mecanismo de construção de núcleos que tenta reunir as vantagens de um núcleo monolítico (desempenho) com as de um micronúcleo (modularidade, portabilidade e uso otimizado de recursos de memória). Um módulo nada mais é que um arquivo objeto o qual pode ser ligado e removido dinamicamente do núcleo do sistema operacional em tempo de execução. Os módulos são compostos por um conjunto de primitivas que implementam uma determinada funcionalidade do sistema operacional como, por exemplo, o sistema de arquivos. A grande diferença em relação à filosofia micronúcleo é que os módulos não são processos separados que se comunicam entre si através de troca de mensagens. Um módulo, após ser carregado e ligado dinamicamente, passa a fazer parte do código do sistema operacional, acessando diretamente estruturas de dados, permitindo assim a comunicação dos módulos com o núcleo através de memória compartilhada. Isso reduz o problema de desempenho apresentado pelos micronúcleos.

O Linux define um conjunto de regras para a definição de interfaces e de estruturas de dados para o desenvolvimento dos módulos. A portabilidade é baseada nesta interface única, pois, por exemplo, um módulo que implementa um *driver* de dispositivo escrito para um disco SCSI pode ser facilmente portado de uma máquina para outra através de uma simples recompilação para a nova plataforma. Além disto o Linux fornece primitivas (chamada de sistema) para o gerenciamento dos módulos que possibilitam a carga e a remoção de módulos em tempo de execução. Assim módulos não utilizados não precisam ser carregados, ou podem ser removidos da memória liberando espaço para outros processos.

9.3 O conceito de processo no Linux

O conceito de processo é fundamental para qualquer sistema operacional multiprogramado. Essencialmente, como visto anteriormente, um processo é uma abstração que representa uma instância de um programa em execução. Os processos podem representar tanto a execução de tarefas do próprio sistema operacional (*deamons*, por exemplo) como tarefas de usuários. É através desse conceito que o sistema operacional organiza suas tarefas de gerenciamento.

Durante o período de vida de um processo, ele utiliza vários recursos do sistema: o processador para sua execução; espaço em memória física para o armazenamento de dados e do programa em execução; descritores de arquivos; o emprego - direta ou indiretamente – de dispositivos de entrada e saída; etc. Para que seja possível gerenciar os recursos do sistema de forma justa, repartindo-os eqüitativamente entre os processos, o Linux deve ter uma idéia clara de o que cada um dos processos existentes no sistema está realizando, ou seja, qual o seu estado de execução. Essa informação é mantida em uma estrutura de dados especial: o descritor de processo. O descritor de processo é uma estrutura bastante complexa, possuindo uma série de campos tais como ponteiros para outros descritores de processos, ponteiros para descritores de arquivos abertos, ponteiros para

áreas de memória em uso, informações para escalonamento, temporizadores, contexto do processo, etc. Nós iremos, na medida do necessário, detalhar alguns desses campos.

9.3.1 O ciclo de vida de um processo: criação

A criação de um processo, em ambientes UNIX, realiza-se da seguinte forma: um processo já existente produz uma cópia exata de si mesmo, duplicando assim todo o seu ambiente através de uma chamada de sistema (`fork`). O novo processo criado é denominado de processo filho, e o processo criador, de processo pai. Em seguida, o processo filho tem sua área de código substituída pelo código que deve ser executado através de outra chamada de sistema: a chamada `exec`. É esta chamada que permite ao processo filho substituir a imagem do código do pai por uma imagem própria. Por isso, encontra-se como uma referência bastante comum, no mundo UNIX, a expressão `fork-exec`. Todos os processos UNIX são criados desta forma.

Na realidade, durante a inicialização de uma máquina, um primeiro processo é criado "manualmente" pela sistema operacional. Esse processo é denominado de `init` e recebe um identificador de processo (*pid*) igual a 1. O processo `init` cria uma série de processos – todos através de `fork-exec` – os quais serão responsáveis pela gerência do sistema. Cada processo criado recebe um identificador diferente (*pid*) que será empregado pelo sistema operacional para referências futuras. Entre os processos criados pelo `init`, estão os processos associados ao programa `getty`. Cada instância do `getty` espera que sejam introduzidos o nome de um usuário e uma senha. Quando essas informações são fornecidas, o processo `getty` substitui seu código, através da chamada `exec`, pelo código do programa `login`. O `login` é responsável por autenticar um usuário, e assim permitir que o mesmo conecte-se ao sistema. O usuário sendo válido no sistema, o `login` é considerado com sucesso e executa novamente uma chamada `exec` para substituir seu código por um outro programa: o interpretador de comandos (`shell`) escolhido pelo usuário (bash, csh, etc.). A partir do `shell`, o usuário pode realizar comandos. Qualquer comando executado no `shell` dará origem a uma seqüência `fork-exec`, permitindo assim a criação de novos processos.

Os comandos executados a partir de um `shell` são normalmente programas utilitários do sistema como, por exemplo, *ls* para listar o conteúdo de um diretório, *grep* para procurar ocorrências de strings em um arquivo, etc. Nesse caso, o `shell` cria um processo filho que executa o código associado a esse comando e faz com que o processo pai (`shell`, nesse caso) espere pelo término do filho. Quando o filho termina, o controle retorna ao `shell` (processo pai) e o usuário pode executar um novo comando. Nesse ponto, cabe um comentário: os processos executados em *background*. Um **processo em *background*** nada mais é que a criação de um processo filho sem que o pai espere por seu término. Dessa forma, podemos ter o processo filho executando ao mesmo tempo que o processo pai.

E para o caso de programas criados por nós usuários? A regra é exatamente a mesma. Ao desenvolver um programa é necessário seguir uma série de passos bem definidos até que se obtenha um programa executável. Um **executável** nada mais é que um formato específico de arquivo determinando áreas para o código, para os dados e para informações de controle necessárias à chamada `exec` permitindo assim ao sistema operacional carregá-lo na memória e em seguida executá-lo. Dessa forma, ao fornecer o nome de um programa executável a partir de uma sessão de `shell`, nós estamos apenas indicando qual código o `exec` deve utilizar na criação do processo filho do processo `shell`. Em outros termos, o nosso programa é filho do `shell`.

Um programa executável em Linux pode possuir vários formatos, mas os formatos comumente utilizados são o formato ELF e o formato `a.out`. Outro tipo de arquivos executáveis são os

shell script. Um ***script*** é uma seqüência de comandos reconhecidos por um `shell` específico. Na execução de um *script*, o `shell` cria um outro processo `shell` para interpretar os comandos do *script*. O `shell` a ser executado é fornecido pelo próprio *script*, por isso a existência (talvez inexplicada ou mágica para você até esse instante) do porquê de todo *script* possuir, na primeira linha, algo do tipo #!/bin/sh. Essa linha está fazendo nada mais que indicar ao `shell` que ele deve executar o programa `sh` (`shell`) como código para o processo filho e utilizá-lo para interpretar os comandos fornecidos no *script*.

Uma última palavra em relação aos procedimentos internos do núcleo na criação de um processo: cada processo, ao ser criado, é associado a um descritor de processos. O procedimento de criação atualiza os diferentes campos do descritor para refletir o estado global atual do processo recém-criado. No Linux, o número máximo de processos é limitado a 255. Isso corresponde ao tamanho do vetor de descritores de processos. Cada entrada desse vetor é associada a um processo.

9.3.2 Ciclo de vida de um processo: execução

Após a criação de um processo, sua execução é iniciada. Durante essa fase, o processo passa por diferentes estados em decorrência de características próprias a sua execução. Um processo Linux pode assumir um dos seguintes estados:

- **TASK_RUNNING**: quando o processo está executando ou esperando para ser executado. Observe que, em relação ao que foi estudado anteriormente, não existem dois estados diferentes para processos prontos para execução (lista de aptos, ou, *ready*) e um outro estado para processo em execução (*running*). O Linux possui uma única lista de processos prontos para execução, e mantém um apontador para um elemento dessa lista para indicar o processo que está atualmente utilizando o processador (o processo *running*). A criação de um processo faz com que seu descritor de processos seja inserido nessa lista.
- **TASK_INTERRUPTIBLE**: é o estado no qual um processo está bloqueado, esperando que uma determinada condição seja satisfeita como, por exemplo, o final de uma operação de entrada e saída, a liberação de um recurso de sincronização (*mutex*, semáforo, por exemplo), ou uma interrupção de software (*signal*) emitida por um outro processo. A condição, ao ser satifeita, faz com o que processo passe para o estado TASK_RUNNING.
- **TASK_UNINTERRUPTIBLE**: corresponde também a um estado bloqueado, porém, nesse caso, o processo está esperando por uma condição crítica – normalmente um evento vinculado diretamente ao hardware – e não pode ter seu estado alterado até que esse evento seja finalizado.
- **TASK_STOPPED**: nesse estado, o processo tem sua execução parada pela ocorrência de certas interrupções de software (*signal*). Um processo STOPPED só continua sua execução após receber outra interrupção de software emitida por um outro processo. Esse comportamento é típico de depuradores, nos quais um processo monitora outro.
- **TASK_ZOMBIE:** estado que um processo filho assume ao terminar enquanto espera a execução, por parte do processo pai, de uma chamada de sistema do tipo `wait`. Através dessa chamada o processo obtém informações relacionadas com o término do filho (lembre-se que em UNIX todos os processos mantém uma relação de filiação, sendo que o pai de todos é o processo `init`). Após a recuperação, por parte do pai, das informações de término do filho, o descritor de processos do filho é liberado.

O controle de passagem de um estado a outro faz parte do procedimento de escalonamento de processos. Nós estudaremos mais detalhadamente o escalonamento do Linux na Seção 9.4.

9.3.3 Ciclo de vida de um processo: término

Um processo notifica o núcleo do sistema operacional do término de sua execução. A forma usual de sinalizar o término é através da chamada de sistema `exit()`. Essa chamada é realizada de forma automática sempre que o último comando de um programa é executado, ou, ainda, ela pode ser colocada explicitamente por programadores em qualquer ponto do programa.

Ao executar `exit()`, o sistema operacional realiza uma série de procedimentos relacionados ao processo que está terminando. Nesse instante, o sistema operacional libera todos os recursos que o processo alocou para si, liberando memória, encerrando arquivos abertos e tornando disponível o descritor de processos. Uma outra forma de terminar a execução de um processo é através da chamada de sistema `kill()`.

9.3.4 Uma palavra sobre threads

Um conceito atualmente bastante em voga em sistemas operacionais é o conceito de *threads*. Esse interesse por *threads* está associado com o advento de máquinas multiprocessadoreas (SMP) e com a facilidade de exprimir atividades concorrentes. Uma ***thread*** é usualmente definida como um fluxo de controle no interior de um processo.

Quando um novo processo é criado, como nós já vimos, o núcleo do Linux copia os atributos do processo corrente para o que está sendo criado. É o procedimento de `fork-exec`. O Linux, entretanto, prevê uma segunda forma de criação de processos: a clonagem. Um processo clone compartilha os recursos (arquivos abertos, memória virtual, etc.) com o processo original. Quando dois ou mais processos compartilham as mesmas estruturas, eles atuam como se fossem diferentes *threads* no interior de um único processo. O Linux não diferencia as estrutura de dados de *threads* e de processos, e, por conseqüência, ambos são tratados indistintivamente por todos os mecanismos de gerência do núcleo. Essa característica é mais visível no escalonamento: *threads* e processos são tratados da mesma forma. A vantagem de criar *threads* está associada ao seu custo de criação (tempo): elas são criadas mais rapidamente que processos pois não necessitam copiar os atributos do processo original, basta inicializar ponteiros de seu descritor de processos de forma que eles referenciem as áreas já existentes do processo que está sendo clonado. Como as *threads* do Linux são reconhecidas pelo núcleo, elas se enquadram no modelo de *threads* 1:1, conforme a classificação apresentada no capítulo sobre programação concorrente.

9.4 Escalonamento em Linux

O problema básico de escalonamento em sistemas operacionais é como satisfazer simultaneamente objetivos conflitantes: tempo de resposta rápido, bom *throughput* para processos *background*, evitar postergação indefinida, conciliar processos de alta prioridade com de baixa prioridade, etc. O conjunto de regras utilizado para determinar como, quando e qual processo deverá ser executado é conhecido como política de escalonamento. Nesta seção, nós estudaremos como o Linux implementa seu escalonador e qual a política empregada para determinar quais processos recebem o processador.

Tradicionalmente, os processos são divididos em três grandes classes: processos interativos, processos *batch* e processos tempo real. Em cada classe, os processos podem ser ainda subdivididos em *I/O bound* ou *CPU bound* de acordo com a proporção de tempo que ficam esperando por operações de entrada e saída ou utilizando o processador. O escalonador do Linux não distingue processos interativos de processos *batch*, diferenciando-os apenas dos processos tempo real. Como todos os outros escalonadores UNIX, o escalonador Linux privilegia os processos *I/O bound* em relação aos *CPU bound* de forma a oferecer um melhor tempo de resposta às aplicações interativas.

O escalonador do Linux é baseado em *time-sharing*, ou seja, o tempo do processador é dividido em fatias de tempo (*quantum*) as quais são alocadas aos processos. Se, durante a execução de um processo, o *quantum* é esgotado, um novo processo é selecionado para execução, provocando então uma troca de contexto. Esse procedimento é completamente transparente ao processo e baseia-se em interrupções de tempo. Esse comportamento confere ao Linux um escalonamento do tipo preemptivo.

O algoritmo de escalonamento do Linux divide o tempo de processamento em épocas (*epochs*). Cada processo, no momento de sua criação, recebe um *quantum* calculado no início de uma época. Diferentes processos podem possuir diferentes valores de *quantum*. O valor do *quantum* corresponde à duração da época, e essa, por sua vez, é um múltiplo de 10 ms inferior a 100 ms.

Outra característica do escalonador Linux é a existência de prioridades dinâmicas. O escalonador do Linux monitora o comportamento de um processo e ajusta dinamicamente sua prioridade, visando a equalizar o uso do processador entre os processos. Processos que recentemente ocuparam o processador durante um período de tempo considerado "longo" têm sua prioridade reduzida. De forma análoga, aqueles que estão há muito tempo sem executar recebem um aumento na sua prioridade, sendo então beneficiados em novas operações de escalonamento.

Na realidade, o sistema Linux trabalha com dois tipos de prioridades: estática e dinâmica. As prioridades estáticas são utilizadas exclusivamente por processos de tempo real correspondendo a valores na faixa de 1-99. Nesse caso, a prioridade do processo tempo real é definida pelo usuário e não é modificada pelo escalonador. Somente usuários com privilégios especiais têm a capacidade de criar e definir processos tempo real. O esquema de prioridades dinâmicas é aplicado aos processos interativos e *batch*. Aqui, a prioridade é calculada, considerando-se a prioridade base do processo e a quantidade de tempo restante em seu *quantum*.

O escalonador do Linux executa os processos de prioridade dinâmica apenas quando não há processos de tempo real. Em outros termos, os processos de prioridade estática recebem uma prioridade maior que os processos de prioridade dinâmica. Para selecionar um processo para execução, o escalonador do Linux prevê três políticas diferentes :

- **SCHED_FIFO**: Essa política é valida apenas para os processos de tempo real. Na criação, o descritor do processo é inserido no final da fila correspondente à sua prioridade. Nessa política, quando um processo é alocado ao processador, ele executa até que uma de três situações ocorra: (i) um processo de tempo real de prioridade superior torna-se apto a executar; (ii) o processo libera espontaneamente o processador para processos de prioridade igual à sua; (iii) o processo termina, ou bloqueia-se, em uma operação de entrada e saída ou de sincronização.

- **SCHED_RR**: Na criação, o descritor do processo é inserido no final da fila correspondente à sua prioridade. Quando um processo é alocado ao processador, ele executa até que uma de quatro situações ocorra: (i) seu período de execução (*quantum*) tenha se esgotado nesse caso o processo é inserido no final de sua fila de prioridade; (ii) um processo de prioridade superior torna-se apto a executar; (iii) o processo libera espontaneamente o processador para processos de prioridade igual a sua; (iv) o processo termina, ou bloqueia-se, em uma operação de entrada e saída ou de sincronização. Essa política também só é válida para processos de tempo real.

- **SCHED_OTHER**: Corresponde a um esquema de filas multinível de prioridades dinâmicas com *timesharing*. Os processo interativos e *batch* recaem nessa categoria.

A criação de processos em Linux, como em todos os sistemas UNIX, é baseada em uma operação do tipo `fork-exec`, ou seja, um processo cria uma cópia sua (`fork`) e em seguida substitui o seu código por um outro (`exec`). No momento da criação, o processo pai (o que fez o `fork`) cede metade de seu *quantum* restante ao processo filho. Esse procedimento é, na verdade, uma espécie de proteção que o sistema faz para evitar que um usuário, a partir de um processo pai, crie um processo filho que execute o mesmo código do pai. Sem essa proteção, a cada criação o filho receberia um novo *quantum* integral. Da mesma forma, o núcleo Linux previne-se contra o fato de um mesmo usuário lançar vários processos em *background*, ou executar diferentes sessões `shell`.

O escalonador do Linux é executado a partir de duas formas diferentes. A primeira é a forma direta através de uma chamada explícita à rotina que implementa o escalonador. Essa é a maneira utilizada pelo núcleo do Linux quando, por exemplo, detecta que um processo deverá ser bloqueado em decorrência de uma operação de entrada e saída ou de sincronização. A segunda forma, denominada de *lazy*, também é conseqüência do procedimento de escalonamento, ocorrendo tipicamente em uma de três situações.

A primeira dessas situações é a rotina de tratamento de interrupção de tempo que atualiza os temporizadores e realiza a contabilização de tempo por processo. Essa rotina, ao detectar que um processo esgotou seu *quantum* de execução aciona o escalonador para que seja efetuada uma troca de processo. A segunda situação ocorre quando um processo de mais alta prioridade é desbloqueado pela ocorrência do evento que esperava. A parte do código que efetua o desbloqueio, isto é, trata os eventos de sincronização e de entrada e saída, consulta a prioridade do processo atualmente em execução e compara-a com a do processo que será desbloqueado. Se o processo a ser desbloqueado possuir uma prioridade mais alta, o escalonador é acionado, e ocorre uma troca de processo. A terceira, e última forma de execução *lazy*, é quando um processo explicitamente invoca o escalonador através de uma chamada de sistema do tipo *yield*. Essa chamada de sistema permite a um processo "passar sua vez de execução" a outro processo, e, para isso, parece claro, é necessário executar o escalonador.

9.5 Gerência de Memória

O subsistema de gerência de memória constitui uma das partes mais importantes e críticas dos sistemas operacionais em geral. A necessidade de utilização de mais memória que a fisicamente disponível em uma máquina tornou-se uma constante no projeto de sistemas operacionais, e uma série de métodos foram criados para solucionar esse problema. A memória virtual é sem dúvida o melhor. Essa técnica baseia-se, essencialmente, na tradução de endereços lógicos em endereços físicos auxiliada por mecanismos implementados no hardware do próprio processador. Os dois mecanismos básicos de tradução são a paginação e a segmentação.

Embora alguns processadores, entre eles os da família Intel, suportem a segmentação, o Linux utiliza muito pouco. Entre as razões pelas quais o Linux não explora a segmentação, estão: a

gerência da paginação é mais simples que a da segmentação; nem sempre o hardware dos processadores oferecem bom suporte à segmentação; e, além disso, normalmente é possível transformar a segmentação em paginação, mapeando-se todo o espaço virtual em um único segmento. Nas próximas seções, nós iremos analisar mais detalhadamente o modelo de memória empregado pelo Linux e seus mecanismos associados.

9.5.1 Memória virtual

Um processo UNIX possui um modelo de memória organizado em quatro partes: texto, dados não inicializados, dados inicializados e pilha. A área de texto corresponde ao código do programa. A área de dados – inicializados ou não - é composta pelo espaço de armazenamento necessário às variáveis alocadas estaticamente no programa. A área de pilha fornece o espaço de memória necessário às variáveis automáticas (locais), para a passagem de parâmetros, e ainda, para salvar e restaurar endereços de retorno dando suporte a sub-rotinas. As áreas de texto e de dados ocupam a parte baixa de memória, e a pilha, a parte superior (a pilha cresce em direção aos endereços inferiores). O espaço existente entre a pilha e os dados é denominado de *heap*. É a partir dessa área que se obtêm porções de memória alocadas dinamicamente.

Nesse modelo, o espaço de endereçamento de um processo qualquer é de 4 Gigabytes, o que corresponde à capacidade total de endereçamento dos processadores de 32 bits (2^{32}). Como a grande maioria das máquinas não possui esse espaço disponível em memória RAM, esse modelo corresponde, na realidade, a uma memória virtual do processo. De forma análoga, seria não produtivo e provocaria um grande desperdício armazenar completamente os processos em disco, pois qualquer processo, independentemente de sua complexidade, necessitaria de 4 Gigabytes de espaço de armazenamento. O conceito de programa executável surge como solução a esses problemas.

Um executável, similarmente ao modelo de processo, é composto por uma área de código, uma área de dados e um conjunto de informações necessárias para carregar esse código executável na mémoria. O executável possui um formato interno que descreve os tamanhos de cada uma das partes do processo, não necessitando mais um espaço de armazenamento idêntico ao tamanho do processo, isto é, 4 Gigabytes. O problema agora consiste em mapear o executável no modelo de memória de processo, e esse, por sua vez na memória física da máquina.

Uma outra questão surge: em um determinado período de tempo o processo não necessita acessar todo o seu código, nem todos os seus dados. Por conseqüência, manter todo o código, ou todos os dados, em memória é um desperdício de espaço de memória. Se considerarmos que o Linux, como todos os sistemas UNIX, é um sistema multiprogramado, nós teríamos esse desperdício multiplicado pelo número de processos em execução. Como a gerência de memória do Linux é baseada em paginação, a memória virtual de um processo é dividida em páginas, e a esse problema de desperdício é resolvido através da técnica de paginação por demanda.

Na **paginação por demanda**, a gerência de memória do sistema operacional procura manter em memória somente as porções de código, dados e pilha que um processo qualquer está utilizando em um determinado período de tempo. Assim, ao invés de carregar toda a área de código e de dados de um processo, o núcleo do Linux mantém apenas uma estrutura interna que descreve a memória virtual do processo, indicando quais partes estão carregadas na memória e quais partes estão armazenadas em disco. Quando um processo necessita acessar uma parte que não está na memória, é gerada uma exceção (*page fault*) a qual é tratada pelo sistema operacional. Esse tratamento consiste basicamente em carregar a parte necessária (página) na memória. Se porventura não

houver mais espaço disponível na memória RAM, o sistema de gerência de memória realiza um procedimento de substituição de páginas (*swapping* na terminologia do Linux).

9.5.2 Paginação em Sistemas Linux

O Linux emprega um sistema de paginação a três níveis para traduzir endereços virtuais (páginas lógicas) para endereços reais (páginas físicas, ou *frames*). Nesse modelo, três tipos de tabelas de páginas são empregadas: diretório global de páginas, diretório intermediário de páginas e a tabela de páginas propriamente dita. Cada uma dessas tabelas contém entradas que apontam para uma tabela do próximo nível hierárquico. Assim, uma entrada do diretório global de páginas aponta para um diretório intermediário de páginas, e, por sua vez, as entradas desse apontam para diferentes tabelas de páginas. A localização (endereço) da tabela que compõem o diretório global de páginas é fornecida por um apontador à parte normalmente implementado por um registrador especial do processador.

A Figura 9.1 ilustra como um endereço virtual trabalha com esse modelo de paginação a três níveis. Inicialmente, é lido o conteúdo de uma entrada do diretório global de páginas. A entrada a ser acessada é obtida, indexando-se essa tabela com o valor do campo *Nível_1*. O contéudo dessa entrada fornece o índice a ser utilizado para acessar uma das entradas da tabela do segundo nível hierárquico, ou seja, um diretório intermediário de páginas. Essa, de forma análoga, utiliza o valor do campo *Nível_2* para fornecer um índice o qual é empregado para fornecer o endereço inicial de uma tabela de páginas. O campo *Nível_3* é então empregado para selecionar uma entrada da tabela de páginas. Entre as informações mantidas na tabela de páginas, está o endereço da página física correspondente a esse endereço virtual. Soma-se, então, a esse o valor correspondente ao campo *deslocamento*. O resultado final é a posição de memória referenciada pelo endereço virtual.

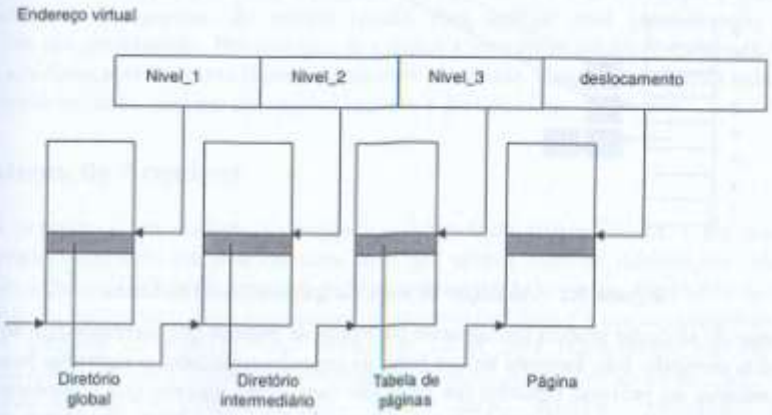

Figura 9.1 – Paginação a três níveis

Um motivo pelo qual o Linux emprega um esquema de paginação a três níveis é seu objetivo de portabilidade a diferentes plataformas. Processadores de 64 bits, como por exemplo os processadores Alpha, normalmente são projetados para realizar uma tradução de endereço lógico em endereço físico empregando três níveis de indireção. Já nos processadores de 32 bits – como a família INTEL 80x86 – o mecanismo de tradução implementado pelo hardware considera apenas

dois níveis. Visando a fornecer suporte a ambas as arquiteturas de processadores, o Linux considera sempre uma paginação a três níveis, adaptando-a logicamente para as arquiteturas de processadores que empregam somente dois níveis. Essa adaptação é feita através de macros específicas a cada plataforma (processador) em que o Linux é executado.

9.5.3 Alocação e liberação de memória física

Para alocar e liberar de forma eficiente páginas físicas (*frames*), o Linux utiliza um algoritmo conhecido como *Buddy*. Nesse esquema, a alocação de memória é feita de forma a alocar um bloco de memória composto por uma ou mais páginas físicas. O controle de áreas de memória livres e alocadas é feito através de um vetor (`free_area`). Cada elemento desse vetor é, na realidade, uma lista encadeada que possui informações das páginas livres no sistema organizadas em blocos. O primeiro elemento aponta para blocos que equivalem a uma página física, o segundo, a blocos de duas páginas físicas, o terceiro, a blocos de quatro páginas físicas, assim crescendo sucessivamente, em potência de dois.

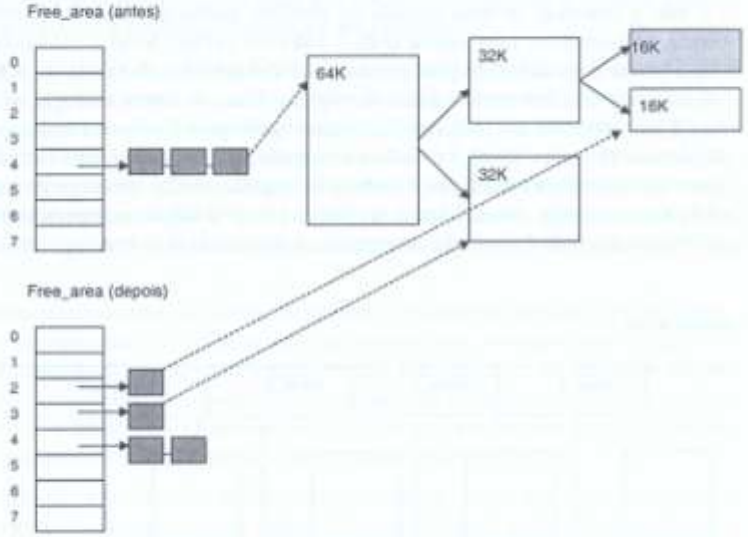

Figura 9.2 – Alocação de memória (página física) no Linux.

O algoritmo de alocação procura inicialmente por blocos de páginas que correspondam ao tamanho de memória desejado. Não havendo blocos livres do tamanho solicitado, o algoritmo busca blocos correspondentes ao próximo tamanho (os quais são duas vezes maiores que o necessário). Esse processo continua até que um bloco livre seja encontrado. Quando o bloco de páginas encontrado for maior que o espaço necessário, ele é subdividido em dois. Esse procedimento é repetido em uma das metades até que se atinja um bloco de páginas no tamanho necessário. Como os blocos são compostos sempre por 2^n páginas, dividi-los em dois implica obter dois blocos de 2^{n-1} páginas cada um. Como apenas um bloco será utilizado, ou subdividido, o outro bloco será inserido na lista mantida pelo vetor `free_area` na entrada correspondente ao seu novo tamanho.

A Figura 9.2 ilustra esse procedimento, considerando a necessidade de uma alocação de 16 Kbytes. Nesse exemplo, considera-se que `free_area` possui entradas livres apenas para três blocos de 64 Kbytes. O algoritmo de alocação particiona um desses blocos de 64 Kbytes em dois blocos de 32 Kbytes. Um dos blocos é inserido no vetor `free_area` na entrada correspondente a 32 Kbytes, o outro bloco é subdividido novamente em dois blocos de 16 Kbytes cada. Novamente, um dos blocos é inserido no vetor `free_area`, o outro bloco é empregado para satisfazer a alocação solicitada.

A alocação de memória tende a transformar os grandes blocos de memória em blocos menores, provocando uma espécie de fragmentação. Para evitar essa fragmentação, o algoritmo de liberação de memória executa o procedimento inverso ao da alocação. Sempre que um bloco de páginas é liberado, o algoritmo verifica se existe um bloco livre de mesmo tamanho (*buddy block*) adjacente ao bloco que está sendo liberado. Se existir, os dois blocos são combinados de forma a criar um único bloco com um tamanho igual a duas vezes o da área que está sendo liberada. Esse procedimento de obtenção de blocos maiores é realizado recursivamente até que os blocos não possam mais formar um único bloco maior.

9.5.4 Swapping

Se, durante a execução de um processo, for necessário carregar uma nova página em memória, e não houver mais espaço disponível, o sistema operacional deve "fazer" espaço em memória substituindo uma página que está em memória pela página que necessita ser carregada. Esse procedimento é conhecido, no Linux, por ***swapping***. Na realidade, o uso do termo *swapping* é um pouco abusivo – embora comumente empregado – já que o mecanismo de substituição é baseado em páginas. O termo correto a ser empregado é ***pagging***, e o movimento das páginas, denominado de *page in* e *page out*. Esse procedimento é realizado por um processo interno do Linux, o *daemon pager*.

O algoritmo utilizado para selecionar a página a ser substituída é o *Least Recently Used* (LRU) combinado com o algoritmo de relógio (*clock*). Para agilizar esse procedimento, algumas otimizações são consideradas. Por exemplo, se a página a ser substituída não foi alterada, ela não é reescrita em disco, pois o mesmo já possui uma cópia atualizada. Caso a página tenha sido alterada, ela é reescrita em disco, mas em um arquivo especial: a área de *swap*.

9.6 Sistema de Arquivos

O Linux organiza o seu sistema de arquivos – como todo sistema UNIX - em uma árvore hierarquizada, resultando em uma estrutura única que agrega todas as informações relativas ao sistema de arquivos. Cada nodo da árvore pode representar um arquivo, um dispositivo de entrada e saída, ou ainda um diretório. Uma importante característica do Linux é a sua capacidade de suportar diferentes sistemas de arquivos. Isso confere ao Linux uma flexibilidade bastante grande, permitindo que ele coexista com outros sistemas operacionais. As versões atuais do Linux suportam 15 sistemas de arquivos diferentes: `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs` e `ufs`. Dessa lista, um sistema de arquivos é particularmente importante: o `ext2`, que é o sistema de arquivos nativo do Linux. Nessa seção, nós analisaremos mais detalhamente a organização desse sistema de arquivos. Antes, porém, nós iremos apresentar alguns conceitos importantes para a compreensão de sistemas de arquivos. O conceito de partições e de pontos de montagem.

9.6.1 Partições e pontos de montagem

Se você já viu uma instalação de Linux, você deve ter-se deparado com a necessidade de criar partições. Uma **partição** nada mais é que a divisão do disco físico em um ou mais discos lógicos aos quais pode ser associado um sistema de arquivos diferente, como por exemplo, uma partição de `swap`, `ext2`, ou ainda `msdos`. A cada sistema de arquivos está associada uma forma de organizar arquivos e diretórios, de controlar o espaço livre e ocupado, etc. Durante a instalação de um sistema, o fato de se criar e inicializar uma partição com um determinado sistema de arquivos faz com que estruturas internas que descrevem a partição sejam inicializadas. O importante a memorizar é o fato de que cada partição pode conter um sistema de arquivos, e que um sistema de arquivos define uma estrutura para o armazenamento de arquivos, de diretórios e de informações de gerência.

O outro conceito importante é o de ponto de montagem. Esse, novamente, é um conceito comum do mundo UNIX. Um **ponto de montagem** nada mais é que um diretório. No UNIX, e no Linux por conseqüência, define-se como ponto inicial de montagem o diretório "/" que corresponde à raiz da árvore que define globalmente o sistema de arquivos. A partir do diretório "/", pode-se definir novos diretórios e montar outros sistemas de arquivos. Montar um sistema de arquivos significa torná-lo parte integrante da árvore que descreve o sistema de arquivos como um todo, disponibilizando-o para acesso.

Observa-se ainda que o sistema de arquivos a ser montado não necessita estar fisicamente no mesmo dispositivo de armazenamento do seu ponto de montagem como é o caso, por exemplo, dos CDROMs. O dispositivo de CDROM possui um sistema de arquivos próprio – `iso9660` – o qual pode ser montado em um ponto qualquer do seu diretório raiz (/). Os administradores de máquinas Linux costumam normalmente montar o cdrom no diretório `/mnt/cdrom`. Após a montagem, o conteúdo do CDROM pode ser acessado normalmente como qualquer outro arquivo. Outros exemplos de montagem de sistemas de arquivos que não residem fisicamente na máquina são os disquetes e arquivos remotos (NFS).

O sistema de gerência de arquivos do Linux, na realidade, possui uma camada a mais entre o sistema de arquivos propriamente dito e o que o usuário "enxerga" como arquivos Linux. Essa camada é o *Virtual File System* (VFS). A função do VFS é isolar o sistema de arquivos real criando, através de uma interface, um sistema de arquivos genérico. Dessa forma, o Linux suporta diferentes sistemas de arquivos, fornecendo ao usuário uma interface única, comum a todos os sistemas de arquivos. Todos os detalhes específicos de cada sistema de arquivos são "escondidos" atrás dessa interface, fazendo com que os sistemas de arquivos sejam "vistos" da mesma forma pelo usuário final. Nós iremos, nas duas próximas seções estudar o sistema de arquivos real do Linux, o `ext2`, para em seguida analisarmos o funcionamento do VFS.

9.6.2 O sistema de arquivos Second Extended File System (`ext2`)

O sistema de arquivos *Second Extended File System*, ou `ext2`, é o sistema de arquivos nativo do Linux, sendo empregado por praticamente todas as suas distribuições desde 1994. Sua estrutura básica é o bloco, que pode armazenar dados ou informações de controle do próprio sistema de arquivos. Uma partição inicializada para conter o sistema de arquivos `ext2` é dividida em um conjunto de grupos blocos e em um bloco que não pertence ao `ext2`, o *boot block*. Esse último é reservado para o procedimento de *boot* da máquina, e dependendo de como o sistema é configurado, é nesse bloco que o LILO, o carregador do Linux (*LInux LOader*), é gravado.

Um arquivo qualquer no Linux é composto por uma quantidade inteira de blocos. Assim, cada arquivo tem seu tamanho arredondado de forma a ocupar um número inteiro de blocos. Por

exemplo, se considerarmos blocos de tamanho de 1024 bytes, um arquivo de 1025 bytes ocupará dois blocos, ou seja, 2048 bytes. Apesar disso, ao verificarmos o tamanho do arquivo através de primitivas do sistema operacional, veremos o seu tamanho "real" de 1025 bytes (essa informação é mantida pela gerência do sistema de arquivos). Essa política de alocação implica que – em média – se desperdice metade do tamanho de um bloco por arquivo. Esse desperdício é, na verdade, resultante de um compromisso entre desempenho e economia de espaço em disco[7].

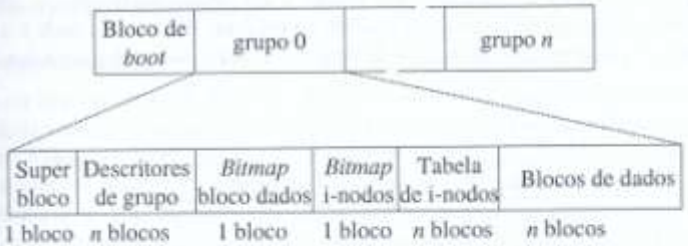

Figura 9.3 – Organização de blocos em uma partição `ext2`.

A organização de blocos da partição em grupos de blocos também está relacionado com um problema de desempenho, tendo como principal objetivo evitar a dispersão de arquivos no disco. Dessa forma, o núcleo tenta alocar para um arquivo – sempre que possível - os blocos pertencentes a um mesmo grupo, reduzindo os movimentos de *seek* no momento de acesso a esse arquivo. A Figura 9.3 mostra essa organização. Cada grupo é composto por seis elementos distintos: super-bloco, descritor do grupo, *bitmap* do bloco de dados, *bitmap* de i-nodos, tabela de i-nodos, e blocos de dados.

- **Super-bloco**: contém uma descrição do tamanho e do formato do sistema de arquivos necessário a sua gerência, uso e manutenção. Normalmente, apenas o super-bloco do grupo 0 é utilizado, porém, por questões de recuperação em caso de falhas, cada grupo de blocos mantém uma cópia do super-bloco. Entre as informações mantidas pelo super-bloco, estão um identificador do sistema de arquivos (número mágico); quantidade de vezes que esse sistema de arquivos já foi montado; número do grupo de bloco que possui essa cópia do super-bloco; o tamanho do bloco; a quantidade de blocos por grupo; o número de blocos livres no sistema; o número de i-nodos livres; e um apontador para o primeiro i-nodo do sistema de arquivos (diretório '/' para o caso de `ext2`).

- **Descritor de grupo**: descreve a estrutura de cada grupo, ou seja, possui tantas entradas quantos grupos existirem. Cada descritor de grupo possui as seguintes informações: o número do bloco no qual está armazenado o *bitmap* que fornece a ocupação dos blocos do grupo; o número do bloco no qual está armazenado o *bitmap* dos i-nodos do grupo; o número do bloco do primeiro i-nodo da tabela de blocos; o número de blocos livres no grupo; o número de i-nodos livres do grupo; e, por último, o número de diretórios que o grupo mantém. Cada grupo possui,

[7] Estão em discussão, e em desenvolvimento, na comunidade Linux, propostas para eliminar esse desperdício de espaço em disco gerenciando a fragmentação interna ao bloco.

também prevendo recuperação em caso de erros, uma cópia dos descritores dos outros grupos.

- **Blocos de dados**: são os diferentes blocos de dados que compõem o espaço de armazenamento de um grupo.

O número de grupos de blocos por partição depende basicamente do tamanho da partição `ext2` e do tamanho definido para o bloco. O fator real limitante do número total de grupos é a restrição que obriga o *bitmap* que fornece a ocupação dos blocos a ser armazenado em um único bloco. Conseqüentemente, cada grupo de bloco deve possuir, no máximo, $8xb$ blocos, onde b é o tamanho do bloco em bytes, e o número total de grupos de blocos é $s/(8xb)$, onde s é o tamanho da partição em blocos.

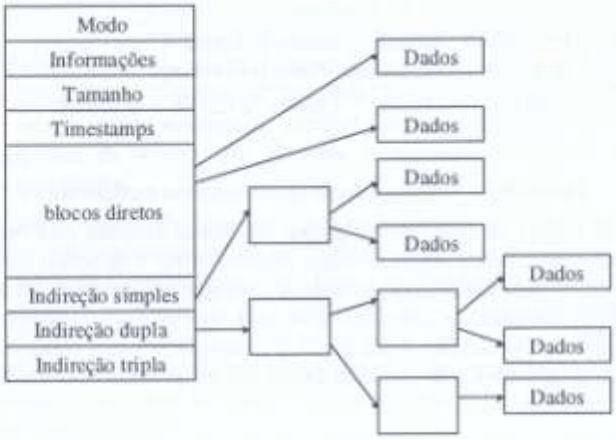

Figura 9.4 – Estrutura de i-nodos do sistema de arquivos `ext2`.

9.6.3 A estrutura de i-nodos do ext2

A estrutura básica de construção de arquivos `ext2` é o i-nodo: cada arquivo ou diretório no sistema de arquivo `ext2` é descrito por um único i-nodo. Cada grupo de blocos, como descrito anteriormente, possui um *bitmap* de i-nodos e uma tabela que contém ponteiros para i-nodos de cada arquivo, ou diretório, pertencente ao grupo. Um i-nodo descreve quais blocos compõem um arquivo e também armazena uma série de informações relativas a esse arquivo. Todos os i-nodos de uma partição `ext2` têm o mesmo tamanho (128 bytes). A Figura 9.4 mostra a organização de um i-nodo:

- **Modo**: esse campo armazena informações relacionadas com direitos de acesso ao objeto que o i-nodo referencia. Cada i-nodo `ext2` pode estar ainda associado com um identificador de tipo como, por exemplo, arquivo regular, diretório, *link* simbólico, dispositivo de E/S, etc;
- **Informações**: fornece os direitos de acesso, leitura e escrita, associando o arquivo a um usuário e/ou a um grupo de usuários;
- **Tamanho:** o tamanho do arquivo em bytes, ou seja, o tamanho real do arquivo e não a quantidade de blocos que ele ocupa.

- ***Timestamps*:** data (hora, dia, mês, ano) em que o i-nodo foi criado ou modificado;
- **Apontadores para blocos**: valores que indicam quais blocos compõem o arquivo. Esses apontadores são empregados em 4 níveis: apontadores diretos (12), apontador de indireção simples (1), apontador de dupla indireção (1), e apontador de tripla indireção (1). Seu funcionamento é o apresentado no capítulo Sistemas de Arquivos. Nas versões atuais do Linux, o tamanho máximo de um arquivo é limitado a 2 Gbytes o que não explora toda a capacidade de endereçamento fornecida pela tripla indireção.

9.6.4 Diretórios ext2

O sistema de arquivo `ext2` implementa diretórios como um tipo de arquivo. O contéudo desse arquivo é uma seqüência de registros, na qual cada registro corresponde a um arquivo pertencente ao diretório. Qualquer diretório, quando criado, é inicializado com dois arquivos com nomes especiais: o arquivo ponto (.) e o arquivo ponto-ponto(..). O primeiro, arquivo ponto, serve para fazer referência ao próprio diretório (diretório corrente), o segundo, arquivo ponto-ponto, referencia o diretório imediatamente superior na hierarquia, ou seja, o diretório pai do diretório corrente. Os diferentes campos que compõem cada um dos registros são:

- **I-nodo**: é o i-nodo que descreve a localização dos blocos que compõem o arquivo;
- **Tamanho da entrada:** tamanho em bytes da entrada do diretório;
- **Tamanho do nome do arquivo:** quantidade de caracteres que formam o nome do arquivo;
- **Tipo do arquivo:** o `ext2` possui uma série de tipos para arquivos (arquivos regulares, *pipes*, diretórios, *links* simbólicos, dispositivos de entrada/saída e *sockets*). Cada um desses arquivos utiliza os blocos de dados de diferentes formas, já que alguns arquivos armazenam dados (arquivos regulares e diretórios) e outros não (dispositivos entrada/saída);
- **Nome:** um string que fornece o nome do arquivo. Esse string possui um caracter especial como marca de fim (\0) e deve sempre ocupar – por questões de desempenho - um tamanho múltiplo de 4 bytes. O tamanho do string é limitado em 256 caracteres.

No `ext2`, um arquivo é localizado através de um caminho (*path*). O caminho é uma seqüência de diretórios separados pelo caractere barra (/) e terminando pelo nome do arquivo. O caminho pode ser relativo ou absoluto. No caminho relativo, o sistema inicia a pesquisa pelo nome do arquivo a partir do diretório corrente. No caminho absoluto, a localização do arquivo é feita a partir do diretório raiz.

Para encontrar o i-nodo que representa um arquivo no sistema `ext2`, o Linux necessita realizar um *parsing* no caminho para encontrar a hierarquia dos diretórios e finalmente acessar o arquivo. O ponto inicial para essa pesquisa é o primeiro i-nodo do super-bloco do sistema, já que nele estão armazenadas as informações relativas ao tamanho e ao formato do sistema de arquivos. De posse dessas informações, é então possível calcular qual grupo de blocos mantém o i-nodo referente a cada uma das entradas do caminho, para em seguida localizar o bloco correspondente no diretório, e acessá-lo para ler o i-nodo do próximo diretório na seqüência do caminho, ou o i-nodo relativo ao próprio arquivo.

9.6.5 O sistema de arquivos Virtual File System (VFS)

Cada vez que um sistema de arquivos é montado, ele é registrado no VFS. Na inicialização da máquina (*boot*), são registrados no VFS todos os sistemas de arquivos definidos em uma tabela de configuração do sistema. Após o *boot*, novos sistemas de arquivos podem ser registrados e eliminados do VFS através de comandos explícitos de montagem e de desmontagem. O fato de registrar um sistema de arquivos no VFS faz com que esse crie uma tabela de i-nodos VFS e uma estrutura de blocos e super-blocos muito similar ao `ext2`.

O super-bloco do VFS mantém uma lista de ponteiros para rotinas que realizam uma operação específica a um sistema de arquivos. Por exemplo, um super-bloco representando um sistema de arquivos do tipo `ext2` possui ponteiros para rotinas de tratamento de i-nodos de acordo com esse formato. A realização de uma operação de leitura fará com que o VFS acione, através de ponteiros para funções, uma rotina específica para a leitura de um arquivo nas normas do `ext2`. De maneira análoga, se o sistema de arquivos montado é MSDOS, nós teremos um super-bloco VFS idêntico ao do caso `ext2`, porém os ponteiros fazem agora referência a rotinas específicas da norma MSDOS. Outro exemplo: ao digitarmos *ls* para verificar o conteúdo de um determinado diretório, o que acontece? A execução desse comando listará o conteúdo do i-node VFS correspondente ao diretório atual. Nesse momento, o VFS traduzirá o comando *ls* pela rotina específica do sistema de arquivos do diretório em questão, *dir*, para um sistema de arquivos MSDOS, *ls* para um sistema de arquivos `ext2`, por exemplo.

9.7 Gerência de Entrada e Saída

Um dos objetivos de um sistema operacional é "esconder" dos usuários as peculiaridades de um determinado hardware. O VFS, apresentado na seção anterior, é um exemplo dessa funcionalidade, pois fornece ao usuário uma interface única para manipulação de arquivos independente do sistema de arquivos e de sua implementação específica. A gerência de entrada e saída realiza um papel análogo, ou seja, fornece uma interface uniforme para acesso a um determinado dispositivo independente de sua tecnologia. Um exemplo bastante comum são os discos rígidos. Nós encontramos discos IDE, EIDE, SCSI, entre outros, os quais possuem geometria, funcionalidades e capacidades diferentes. Entretanto, eles são "vistos" de forma genérica como discos rígidos. Esse é o objetivo da gerência de entrada e saída: criar uma camada de software que esconda, através de uma interface comum, os detalhes específicos de cada dispositivo. Como isso é feito?

Na verdade, o processador não é o único dispositivo "inteligente" de um hardware. Cada dispositivo físico de entrada e saída possui um controlador. Um controlador nada mais é que um processador projetado para realizar uma determinada função específica. Assim, teclado, mouse e portas seriais são normalmente vinculados a um controlador serial; discos IDE, a um controlador IDE; discos SCSI, a um controlador SCSI; interface de rede, a controladores de redes; etc. Cada controlador possui um conjunto próprio de registradores de controle e estado (CSR – *Control and Status Registers*) os quais diferem de controlador para controlador. Os CRS são utilizados para inicializar, encerrar, diagnosticar e realizar operações em um controlador, ou seja, em um dispositivo. Ao invés de cada aplicação codificar a seqüência de comandos CSR, esses estão concentrados no núcleo do sistema operacional já que, por exemplo, ler e escrever são operações comuns a todas as aplicações. O software que implementa, no núcleo do sistema operacional, essas operações é conhecido como *driver* de dispositivos (*device driver*).

9.7.1 *Drivers* de dispositivos (Device drivers)

O sistema operacional interage com os dispositivos de entrada e saída através dos *drivers* de dispositivos (*device drivers*). Um *driver* de dispositivo é composto por um conjunto de funções e de estruturas de dados que controlam um ou mais periféricos tais como discos rígidos, teclados, mouses, monitores, interfaces de rede, etc. Cada *driver* interage com as diferentes partes do sistema operacional através de uma interface de programação (API) bem definida.

A Figura 9.5 ilustra como um *driver* de dispositivo interage com o núcleo e com os processos de usuário. Um processo *P*, de um usuário qualquer, ao desejar realizar uma operação em um periférico de entrada e saída envia uma requisição ao núcleo através de primitivas de manipulação de arquivos. Esse procedimento é conseqüência direta do fato de que em UNIX, cada dispositivo de entrada e saída é visto como um arquivo especial normalmente encontrado no diretório `/dev`. Na prática, cada um desses arquivos especiais referencia um periférico de entrada e saída de um tipo específico de *driver* de dispositivo.

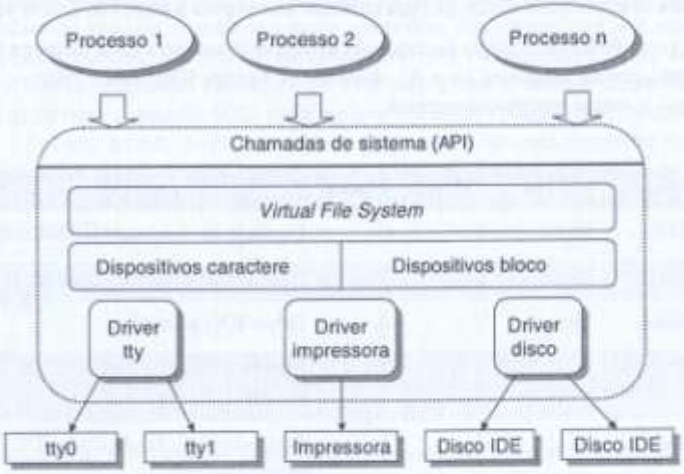

Figura 9.5 – Organização em camadas da gerência de entrada e saída.

Nós vimos precedentemente que o VFS utiliza um conjunto canônico de operações (*open, read, lseek, etc.*) para acessar diferentes sistemas de arquivos. Já que os dispositivos de entrada e saída também correspondem a arquivos, nós podemos imaginar um tipo de "sistema de arquivos" específico ao tratamento dos periféricos de entrada e saída. Dessa forma, é possível empregar as mesmas operações canônicas do VFS para acessar periféricos. Nesse caso, a implementação real de cada uma das funções canônicas fica a cargo do *driver* do dispositivo. Como cada *driver* de dispositivo está associado a um controlador de dispositivo, e esse, por sua vez, possui comandos e *status* específicos, a maior parte dos dispositivos de entrada e saída possuem seu próprio *driver*.

9.7.2 Arquivos de dispositivos

Os arquivos de dispositivos são os arquivos especiais empregados para representar os dispositivos de entrada e saída. Além de um nome, cada um desses arquivos possui três atributos: tipo, número maior, número menor. O tipo do dispositivo indica se ele é um dispositivo orientado a caractere ou a bloco (nós voltaremos a esse ponto mais tarde). O número maior (*major number*) é um número na faixa de 1 a 255 que serve para identificar uma classe de dispositivos. Todos os arquivos de dispositivos que possuem o mesmo número maior e o mesmo tipo dividem um mesmo *device driver*, pois isso representa que eles são da mesma classe. O número menor (*minor number*) identifica um dispositivo específico entre um grupo de dispositivos de uma mesma categoria, ou seja, de um mesmo *major number*.

Normalmente, cada arquivo de dispositivo é associado a um dispositivo físico de entrada e saída, ou a uma partição lógica de um disco. Em alguns casos, entretanto, um arquivo de dispositivo pode não estar associado a um periférico físico mas sim a um dispositivo lógico fictício. É o caso, por exemplo, do dispositivo "/dev/null" que corresponde a um "buraco negro": todos os dados escritos nesse dispositivo são sempre descartados. Esse artifício é empregado por programadores mais experientes para desconsiderar dados ou para eliminar mensagens padrões de certos softwares.

A Tabela 9.1 exemplifica o conceito de arquivos de dispositivos para uma máquina hipotética que possui duas unidades de disquetes (A e B), dois discos rígidos IDE (hda e hdb) – cada um com duas partições – e outros periféricos comuns.

Nome	Tipo	Major	Minor	Descrição
/dev/fd0	bloco	2	0	Unidade de disquete 0 (driver A)
/dev/fd1	bloco	2	1	Unidade de disquete 1 (driver B)
/dev/hda	bloco	3	0	Disco IDE primário
/dev/hda1	bloco	3	1	Primeira partição do disco IDE primário
/dev/hda2	bloco	3	2	Segunda partição do disco IDE primário
/dev/hdb	bloco	3	64	Segundo disco IDE
/dev/hdb1	bloco	3	65	Primeira partição do disco IDE secundário
/dev/tty0	caractere	3	0	terminal
/dev/lp1	caractere	4	1	Impressora paralela
/dev/console	caractere	1	1	teclado

Tabela 9.1 – Exemplos de arquivos de dispositivo.

9.7.3 Dispositivos orientados a caractere, orientados a bloco e de rede

O Linux classifica os dispositivos de entrada e saída em três categorias: orientados a caractere, orientados a bloco e de rede. Os dispositivos orientados a caractere são aqueles que permitem a transferência de uma quantidade arbitrária de dados em uma única operação de entrada e saída, variando desde uma escrita byte a byte até a de uma seqüência de tamanho qualquer. A maior parte dos dispositivos de entrada e saída recaem nessa classe. Os dispositivos orientados a bloco, por sua vez, estão associados aos dispositivos que permitem o acesso aleatório a dados, tipicamente, os discos rígidos, unidades de disquetes, CDROM, etc. Os dispositivos orientados a blocos são capazes de transferir apenas uma quantidade fixa de dados (o bloco) em uma única operação de entrada e saída.

Tanto os dispositivos orientados a caractere como os orientados a blocos são associados a arquivos de dispositivos e são acessados através de funções canônicas do VFS. Entretanto, isso não é verdade para um tipo de dispositivo de entrada e saída: as interfaces de rede.

As facilidades de rede, como conhecemos atualmente, foram inicialmente inseridas na família BSD do UNIX. A solução adotada para o envio e recepção de dados de sistemas remotos foi a de atribuir para cada interface de rede do sistema um nome simbólico. Esse nome não tem correspondência com nenhum i-nodo, ou com um arquivo de dispositivo. Por conseqüência, nessa solução, não é mais possível o uso de primitivas canônicas do VFS para enviar e receber dados pela rede. Para contornar esse problema, a solução BSD contemplava um outro conjunto de chamadas de sistema (por exemplo, `listen`, `bind`, `socket`, etc.) as quais ofereciam uma abstração de endereço de rede para identificar os pares de uma comunicação. A solução BSD e esse grupo de chamadas de sistemas tornaram-se um padrão de programação para dispositivos de interconexão de redes. O Linux segue essa filosofia.

9.8 Exercícios

1) O Linux não permite que páginas pertencentes ao núcleo sofram o mesmo procedimento de paginação que os processos de usuários. Por que essa restrição? No seu ponto de vista, quais as vantagens e desvantagens de tal abordagem? (Seção 9.5)

2) O código do Linux é disponível gratuitamente através da Internet em vários sites. Qual a conseqüência dessa disponbilidade do código para a segurança e o desenvolvimento do Linux? (Seção 9.1)

3) Compare o procedimento da chamada de sistema `clone` com os diferentes métodos existentes para a implementação de *threads* (modelo 1:1, 1:N, M:N). (Seção 9.3)

4) Analise as vantagens e desvantagens do sistema de módulos carregáveis do Linux em relação às filosofias de núcleo monolítico e micronúcleo. Aborde aspectos como desempenho, configuração do sistema, atualizações e portabilidade. (Seção 9.2)

10
Windows 2000

O Windows NT é um sistema operacional proprietário, desenvolvido pela Microsoft, que surgiu com o objetivo ser um sistema operacional da década de 90, atento aos novos avanços tecnológicos e às exigências do mercado. Desde o seu lançamento, em 1993, com a versão 3.1, o Windows NT teve a preocupação de fornecer suporte a ambientes de rede e distribuídos. Sua evolução, até chegar ao Windows NT 5.0, comercialmente conhecido como Windows 2000, foi orientada à interoperabilidade com outros sistemas operacionais. A estrutura básica da arquitetura Windows NT manteve-se praticamente inalterada desde o seu lançamento até as versões mais recentes. Neste capítulo, nós apresentaremos detalhes da arquitetura do Windows 2000 seguindo, na medida do possível, a mesma ordem de apresentação do capítulo sobre Linux para que o leitor possa ter uma visão comparativa entre esses dois sistemas. Nós concluiremos este capítulo com uma breve apresentação do sucessor do Windows 2000: Windows XP.

10.1 Introdução: um pouco de história

A história do Windows NT iniciou com o desejo da Microsoft de criar um sistema operacional que explorasse as inovações tecnológicas apresentadas pelos processadores no final da década de 80, início da década de 90. O objetivo era desenvolver um sistema operacional multitarefa para ser utilizado tanto em ambientes monousuário como multiusuário. O nome *Windows* é originário de um sistema de janelas (*Windows 3.x for Workgroup*) projetado para competir com a interface usuário dos computadores Macintosh (Apple). Esse ambiente de janelas emprestou a sua "aparência" para a primeira versão do Windows NT. A sigla NT vem de *New Technology*, e foi criada para caracterizar a nova filosofia que orientou a sua concepção.

A primeira versão do Windows NT (versão 3.1) foi lançada em 1993 e constituiu o primeiro sistema operacional de 32 bits da Microsoft. Esse sistema operacional caracterizava-se por fornecer uma compatibilidade com o sistema operacional MS-DOS, com aplicações desenvolvidas para o "velho" sistema de janelas (*Windows 3.x for Workgroup*) e com o sistema operacional OS/2. Na realidade, o sistema OS/2 foi, em conjunto com a IBM, o primeiro esforço da Microsoft em desenvolver um sistema operacional multitarefa em 32 bits. O OS/2, apesar de possuir em sua concepção uma série de inovações e de soluções inteligentes, jamais vingou como sistema operacional por ser demasiado "pesado" para ser executado nas máquinas existentes na época. A experiência de desenvolvimento do OS/2 influenciou fortemente a concepção do Windows NT.

Após sucessivas versões do Windows NT 3.x, nasce o Windows NT 4.0. Em relação à arquitetura interna de sistema operacional, o NT 4.0 mantém essencialmente a mesma de seu predecessor (Windows NT 3.x). As principais modificações em relação ao Windows NT 3.x estão na interface gráfica, que agora se assemelha à do Windows 98, e na migração de vários serviços, também relacionados com a parte gráfica, do subsistema Win32 para o núcleo do Windows NT 4.0.

Em 1999, a Microsoft lançou uma nova versão do Windows NT, a 5.0, que comercialmente recebeu o nome de Windows 2000. A estrutura básica do sistema operacional é a mesma do NT 4.0. A principal diferença está na inclusão de serviços orientados a ambientes distribuídos e de rede. Na realidade, dependendo das funcionalidades adicionadas ao Windows 2000, existem 4 diferentes versões desse sistema operacional:

- Windows 2000 Professional, o qual substitui o NT workstation, isto é, as máquinas empregadas como ponto de trabalho (máquina cliente).
- Windows 2000 Server, equivalente ao NT Server. Essa configuração apresenta alguns serviços orientados ao compartilhamento de recursos e destina-se, como o próprio nome induz, a máquinas servidoras em uma rede NT.
- Windows 2000 Advanced Server, que fornece uma série de facilidades para ambientes de rede e distribuídos, incluindo o conceito de *clustering* (seção 10.9) e suporte ao balanceamento de carga.
- Windows 2000 Datacenter Server, que agrega todas as funcionalidades disponíveis no Windows 2000 e suporta o endereçamento de até 64 GB.

No momento em que preparávamos esta edição, a Microsoft estava prestes a lançar seu novo produto: o Windows *eXPerience*, comercialmente denominado de Windows XP. O Windows XP consiste na geração seguinte da família Windows, e na seção 10.10, nós faremos uma breve apresentação sua.

10.2 Diretrizes de projeto

O desenvolvimento do Windows 2000 foi orientado por cinco objetivos principais que sempre nortearam o projeto de todos os produtos da família Windows NT: confiabilidade e robustez; extensibilidade e facilidade de manutenção; portabilidade; desempenho; e, por último, conformidade com o padrão POSIX e certificação C2.

O objetivo de confiabilidade e robustez traduz-se no fato de que um sistema deve ter a capacidade de se proteger do mau funcionamento e de problemas oriundos do próprio sistema operacional, assim como de fontes externas (ataques). Para atingir essa meta, algumas diretrizes foram traçadas no desenvolvimento do Windows 2000. Inicialmente, o sistema foi primeiro concebido e documentado para somente após começar a ser codificado. Nesse procedimento inicial, elaborou-se uma interface clara e bem definida para os serviços do núcleo com o intuito de evitar o uso de parâmetros "mágicos" e *flags* para a chamada desses serviços. Essa decisão forçou a elaboração de uma documentação clara da interface de serviços, o que facilitou a fase de testes. Componentes importantes do Windows 2000, como, por exemplo, Win32, OS/2, e POSIX, foram isolados em subsistemas, cada um com uma interface de serviços bem definida e com tratamentos de exceção próprios, isolados a esse subsistema. Essa abordagem, na realidade, dividiu a complexidade do sistema em dois grandes grupos: serviços do núcleo e serviços de subsistemas. Um erro em um subsistema afeta apenas o comportamento deste, não do sistema operacional como um todo.

O segundo objetivo de projeto, extensibilidade e facilidade de manutenção, diz respeito à perenidade do sistema. Era necessário que o Windows 2000 soubesse evoluir, adaptando-se facilmente a novas necessidades, tanto de hardware como de software. O suporte a ambientes distribuídos e a filosofia de subsistemas são pontos importantes para o cumprimento dessa meta. A estruturação em subsistemas permite isolar alterações a apenas uma parte do sistema operacional:

se, por exemplo, a norma POSIX for alterada, apenas o subsistema POSIX é diretamente afetado. Essa concepção modular facilita a adição de novos subsistemas ao Windows 2000. O "isolamento" do subsistema através de uma interface de serviços reduz o risco de, ao modificar o sistema operacional, serem introduzidos efeitos colaterais.

Os ambientes de computação distribuída caracterizam-se pelo fato de potencialmente empregarem plataformas heterogêneas. A portabilidade de um sistema torna-se então um objetivo importante para um sistema operacional que almeja ser empregado em tais ambientes. Por portabilidade, entende-se a facilidade que um sistema apresenta em ser recompilado e executado em diferentes plataformas de hardware com o mínimo possível de alterações em seu código.

Os objetivos de confiabilidade e robustez, extensibilidade e facilidade de manutenção, e o de portabilidade levam, de uma certa forma, a uma programação de forma modular, fazendo com que diferentes módulos executem diferentes tarefas disponibilizadas na forma de uma interface de programação (API). Esse modelo de concepção de sistemas normalmente se traduz em sobrecustos (*overheads*) na realização de tarefas do sistema. O desempenho surge, então, como outro objetivo desejável. O projeto do Windows 2000 manifesta essa preocupação em dois aspectos. O primeiro traduz-se em redefinir estruturas de dados e algoritmos de forma a otimizá-los o máximo possível. O segundo aspecto implica que a definição de algoritmos e estruturas de dados não vá contra a filosofia modular de subsistemas e do uso de interfaces claras e bem definidas para os serviços. É necessário definir soluções que contemplem, ao mesmo tempo, a filosofia modular de subsistema e o desempenho, encontrando assim uma boa relação custo-benefício.

O quinto objetivo do Windows 2000 é, na realidade, composto por duas partes: conformidade POSIX e certificação C2. A primeira, conformidade com a norma POSIX, permite que aplicações desenvolvidas para executar em ambientes UNIX sejam facilmente portadas para Windows 2000 e vice-versa. Esse objetivo envolve, de certa forma, projetar as interfaces de serviços (chamadas de sistema) para ter um aspecto UNIX-*like*. Essa decisão tem um impacto importante se considerarmos a série de ferramentas UNIX voltadas a ambientes distribuídos e de rede, além, é claro, de reduzir o número de "versões" de interfaces de programação disponibilizadas por diferentes sistemas operacionais. Outro ponto não negligenciável é o fato de que, cada vez mais, licitações nos Estados Unidos impõem como condição o fato de o sistema operacional possuir conformidade POSIX. Aqui surge também a segunda parte, a certificação C2. Essa certificação diz respeito a normas de segurança, facilidades de auditoria, deteção de ataques, controle de quotas e de acesso a recursos do sistema. Novamente, todas as licitações americanas estão exigindo essa certificação. Então para atender esse requisito de mercado, cada vez mais comum no mundo inteiro, o projeto do Windows 2000 foi orientado a possuir conformidade POSIX e a oferecer os serviços necessários à obtenção da certificação C2.

10.3 Arquitetura do Windows 2000: visão geral

Um sistema operacional é um *software* extremamente complexo. Assim, vários modelos de arquiteturas foram propostos para melhor organizar os detalhes de sua implementação. Esses modelos vão desde sistemas baseados em *kernel* monolítico até sistemas totalmente moduláveis, baseados em micronúcleo (*microkernel*).

A arquitetura do Windows 2000 é fortemente inspirada no princípio de micronúcleo. Assim, cada funcionalidade do sistema é oferecida e gerenciada por um único componente do sistema operacional. Os demais componentes do sistema operacional e todas as aplicações acessam os serviços providos por um determinado componente através de uma interface bem definida.

Teoricamente, cada módulo (componente) pode ser removido, atualizado, ou substituído sem necessitar de alterações nais demais partes do sistema. O Windows 2000 não é puramente orientado à filosofia micronúcleo porque módulos fora do micronúcleo executam operações em modo protegido (modo kernel). A justificativa para essa decisão de projeto está no desempenho. Em uma abordagem orientada a micronúcleo "pura", uma aplicação que necessite executar uma operação privilegiada deve solicitar esse serviço ao micronúcleo. Esse procedimento envolve uma série de trocas de contexto. No Windows 2000, para evitar essa troca de contexto, certos subsistemas (módulos ou componentes) passam de modo usuário para modo protegido e implementam diretamente a função desejada, evitando assim a passsagem pelo núcleo e as trocas de contexto que isso implica.

O Windows 2000 segue também uma organização em camadas. Nessa abordagem, o sistema operacional é dividido em módulos que são dispostos uns sobre os outros em camadas. Cada camada oferece um conjunto de serviços à camada superior e só pode utilizar serviços fornecidos pela camada imediatamente inferior. Outro conceito explorado pelo Windows 2000 é o modelo orientado a objetos. Nesse modelo, recursos do sistema, arquivos, memória e dispositivos físicos, são implementados por objetos e manipulados através de métodos (serviços) associados a esses objetos.

O Windows 2000 foi projetado de forma a permitir a execução de aplicações escritas para outros sistemas operacionais. Essa facilidade é suportada a partir de subsistemas que, implementados como um processo separado, fornecem um ambiente de execução compatível a um determinado sistema operacional. Esse ambiente é composto, além de uma interface gráfica e de um interpretador de comandos, por uma interface de programação (API) compatível com os serviços (chamadas de sistema) do sistema operacional que o subsistema implementa. Isso implica que uma aplicação escrita para um sistema operacional particular pode executar sem alterações no Windows 2000 por "enxergar" as mesmas funções existentes no sistema nativo para o qual foi escrito. O mais importante dos subsistemas do Windows 2000 é o Win32, que possibilita que aplicações escritas para outros sistemas operacionais Microsoft executem no Windows 2000 sem problemas. Outros subsistemas disponíveis são o subsistema OS/2 e o subsistema POSIX.

A estrutura do Windows 2000 pode ser dividida em duas partes: modo usuário (onde estão localizados os subsistemas protegidos) e modo kernel (o executivo). Os subsistemas protegidos são assim denominados porque residem em processos separados cuja memória é protegida do acesso de outros processos. Os subsistemas interagem entre si através de um mecanismo de troca de mensagens (*Local Procedure Call - LPC*). No modo kernel, rodam os componentes do sistema operacional que necessitam de desempenho e por isso interagem com o hardware e um com o outro sem estarem sujeitos a trocas de contexto e de modo. Todos os componentes estão protegidos das aplicações porque estas não possuem acesso à parte protegida do sistema operacional. Ainda, cada componente está protegido um do outro devido à adoção da orientação a objetos. Todo acesso a um objeto é feito através de um método.

O modo kernel é estruturado em três grandes módulos funcionais: *hardware abstraction layer*, drivers de dispositivos e o executivo. A camada denominada de *hardware abstraction layer* (HAL) é um módulo carregável do núcleo. Esse módulo respeita uma interface padrão de serviços, porém possui uma implementação específica para o hardware no qual o Windows 2000 está executando. Todas as funcionalidades que são dependentes de um determinado hardware, como interfaces de E/S, controladores de dispositivos e de interrupções, ou ainda, o próprio processador, são implementadas dentro desse módulo. Esse tipo de projeto permite que todos os componentes do sistema operacional acima do módulo HAL executem de forma independente do hardware, fornecendo assim o grau de portabilidade necessário a um sistema que visa a operar em ambientes heterogêneos.

Os drivers de dispositivos são outra categoria de módulos carregáveis do núcleo. Esses drivers oferecem, dentro do executivo do Windows 2000, uma interface entre o sistema de E/S e o HAL. O gerenciamento dos drivers de dispositivos foi um dos aspectos em que o Windows 2000 (NT 5.0) apresentou uma evolução significativa em relação a sua versão anterior (Windows NT 4.0) através da integração de suporte a *plug-and-play*.

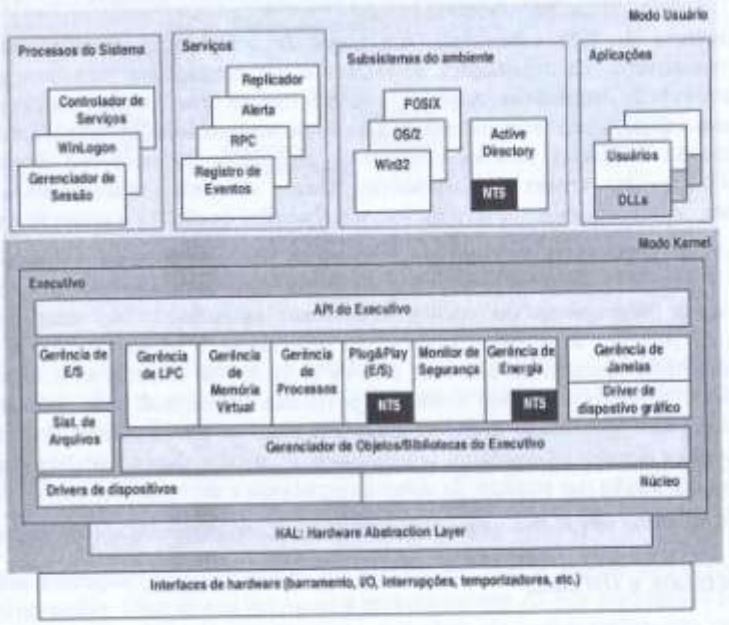

Figura 10.1 – Componentes da estrutura do Windows 2000

O executivo constitui o núcleo do sistema operacional Windows 2000. É ele que implementa os serviços básicos do Windows 2000, exportando funções para serem utilizadas em modo usuário e funções que só são acessíveis por componentes (módulos) pertencentes ao próprio núcleo. Os principais componentes do executivo são (Figura 10.1):

- **Gerência de objetos**: é o componente responsável por criar, gerenciar e excluir objetos do Executivo Windows 2000. Entende-se por objetos do executivo a abstração de todos os tipos de dados utilizados para representar recursos do sistema operacional como processos, *threads*, alocação de memória, mecanismos básicos de sincronização (mutex e semáforos), etc.
- **Gerência de processos e *threads***: responsável por criar, encerrar, suspender e dar prosseguimento à execução de *threads* e processos. Ainda, armazena e recupera informações sobre os processos e *threads* do Windows 2000. A forma de tratamento de *threads* e processos do Windows 2000 será detalhada na seção 10.4.

- **Gerência de memória virtual**: módulo responsável pela implementação do suporte a memória virtual e do gerenciamento de outras atividades relacionadas à gerência de memória como proteção, cache, mapeamentos, etc.
- **Monitor de segurança**: faz cumprir as políticas de segurança no computador local. Verifica acesso aos recursos do sistema operacional, protegendo e auditando os objetos durante sua execução.
- **Módulo de suporte a *Local Procedure Call* (LPC)**: módulo responsável pela comunicação por troca de mensagens entre processos. Esse mecanismo é basicamente uma versão otimizada do conceito de *Remote Procedure Call* (RPC).
- **Gerência de E/S:** compreende um grupo de componentes responsáveis pelo processamento de informações de entrada e por emitir saída para uma grande variedade de dispositivos. A gerência de E/S fornece uma interface padrão para o executivo de forma independente do tipo de dispositivo de E/S. As solicitações de E/S são traduzidas para os dispositivos específicos de *hardware* através da utilização dos drivers de dispositivos. Outros elementos relacionados a esse componente são o sistema de arquivos, o gerenciador de cache e o driver de rede.

Para concluir esta seção, resta comentar que o núcleo do Windows 2000 foi projetado de forma a dar suporte a multiprocessamento simétrico quando executado em máquinas multiprocessadoras. Essa abordagem distingue-se do multiprocessamento assimétrico. No multiprocessamento assimétrico, na presença de n processadores, um processador é pré-selecionado e dedicado à execução do sistema operacional, deixando para os processos de usuários os $n-1$ processadores restantes. Já no multiprocessamento simétrico, o sistema operacional pode ser executado em qualquer processador que esteja livre, ou ainda em todos os processadores simultaneamente, explorando melhor o potencial dos vários processadores existentes. Essa possibilidade tem impacto importante na concepção dos serviços do sistema operacional e deve ser considerada no momento do projeto do sistema.

10.4 Processos e *threads*

Cada sistema operacional tem a sua própria forma de implementar processos; as variações estão nas estruturas de dados utilizadas para representar fluxos de execução, sua denominação, como são protegidos uns em relação aos outros e na forma de inter-relacionamento. O Windows 2000 implementa o conceito de processo a partir de dois objetos: objeto processo e objeto *thread*. O objeto processo é a entidade que corresponde a recursos do sistema tais como memória, arquivos, etc. O objeto *thread*, por sua vez, constitui uma unidade de trabalho que é executada de forma seqüencial e podendo ser interrompida em qualquer ponto.

A criação de um processo em Windows 2000 corresponde a instanciar (criar) um objeto do tipo processo, o qual é uma espécie de "molde" para novos processos. Nesse momento, uma série de atributos são inicializados para esse novo processo, como, por exemplo, um identificador de processo (*pid*), descritores de proteção, prioridades, quotas, etc. A unidade de escalonamento do Windows 2000 é o conceito de *thread*. A cada processo está associada, no mínimo, uma *thread*. Cada *thread* pode criar outras *threads*. Essa organização permite a execução concorrente dos processos, além de possibilitar uma concorrência entre as *threads* que pertencem a um mesmo processo. Uma *thread* pode estar em um de seis estados (Figura 10.2):

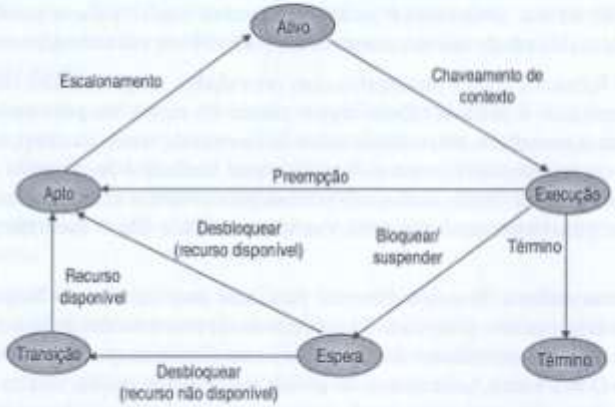

Figura 10.2 – Diagrama de estados de *threads* no Windows 2000

- **Apto** (*Ready*): corresponde ao estado no qual se encontram as *threads* aptas a executar, ou seja, as *threads* que o escalonador considera para selecionar a próxima a ser executada. Uma vez selecionada, a *thread* passa ao estado ativo (*standby*).
- **Ativa** (*Standby*): estado intermediário no qual a *thread* selecionada pelo escalonador espera pelo chaveamento de contexto para entrar efetivamente em execução. No sistema existe, por processador, apenas uma *thread* nesse estado.
- **Em execução** (*running*): estado que assume uma *thread* quando está ocupando o processador. Uma *thread* em *running* executa até que ela seja preemptada por uma *thread* de mais alta prioridade, esgote a sua fatia de tempo, realize uma operação bloqueante, ou termine. Nos dois primeiros casos, o descritor da *thread* é reinserido na lista de aptos (estado *ready*).
- **Espera** (*waiting*): uma *thread* passa a esse estado sempre que (1) for bloqueada pela espera da ocorrência de um evento (e.g. E/S); (2) realizar uma primitiva de sincronização; ou (3) quando um subsistema ordena a suspensão da *thread*. Quando a condição de espera é satisfeita, a *thread* é inserida na lista de aptos.
- **Transição** (*transition*): corresponde ao estado em que uma *thread* está apta a ser executada, porém os recursos de sistema necessários a sua execução (e.g. estar paginada em memória), ainda não estão disponíveis. Quando esses recursos estão disponibilizados, a *thread* passa ao estado apto.
- **Término** (*terminated*): estado que uma *thread* assume quando atinge seu final, ou é terminada por uma outra *thread*, ou ainda quando o processo a que está associada termina.

As *threads* de qualquer processo, inclusive as do executivo do Windows 2000, podem, em máquinas multiprocessadoras, ser executadas em qualquer processador. Dessa forma, o escalonador do Windows 2000 atribui uma *thread* pronta a executar (apta) para o próximo processador disponível. Múltiplas *threads* de um mesmo processo podem estar em execução simultaneamente.

O escalonador do Windows 2000 é preemptivo com prioridades. As prioridades são organizadas em duas classes: tempo real e variável. Cada classe possui 16 níveis de prioridades, sendo que as *threads* da classe tempo real têm precedência sobre as *threads* da classe variável, isto é, sempre que não houver processador disponível, uma *thread* de classe variável é preemptada em favor de uma *thread* da classe tempo real. Todas as *threads* prontas para executar são mantidas em estruturas de filas associadas a prioridades em cada uma das classes. Cada fila é atendida por uma política *Round-robin*.

A atribuição de prioridades a *threads* é diferente para cada uma das classes. Enquanto na classe de tempo real, as *threads* possuem prioridade fixa, determinada no momento de sua criação, as *threads* da classe variável têm suas prioridades atribuídas de forma dinâmica (por isso o nome de "variável" para essa classe). Dessa forma, uma *thread* de tempo real, quando criada, recebe uma prioridade e será sempre inserida na fila dessa prioridade, ao passo que uma *thread* da classe variável poderá migrar entre as diferentes filas de prioridades. Em outros termos, o Windows 2000 implementa um esquema de múltiplas filas para as *threads* da classe real e múltiplas filas com realimentação para as *threads* da classe variável. A Figura 10.3 ilustra o sistema de prioridades do Windows 2000.

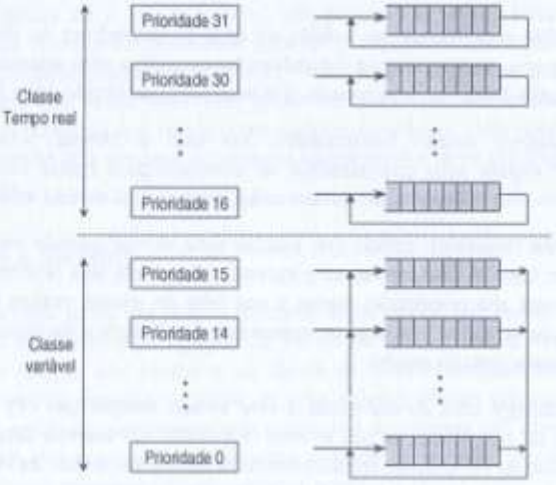

Figura 10.3 – O esquema de prioridade do Windows 2000

Na classe variável, a prioridade de uma *thread* é estabelecida a partir de dois parâmetros, um vinculado à própria *thread* e outro, ao processo a que pertence. Um objeto processo, durante sua criação, recebe um valor, entre zero e 15 inclusive, para sua prioridade de base. Cada *thread* recebe uma prioridade inicial, variando 2 unidades acima ou abaixo da prioridade de base do processo, que indica sua prioridade relativa dentro desse processo. A prioridade de uma *thread* varia durante a sua vida, mas nunca assumirá valores inferiores a sua prioridade base nem superiores a 15. O critério

empregado para variar a prioridade de uma *thread* é o tempo de utilização do processador. Se a *thread* for preemptada por ter executado durante todo o quantum de tempo que lhe foi atribuído, o escalonador do Windows 2000 diminui sua prioridade; caso contrário, sua prioridade é aumentada. Em outros termos, o escalonador do Windows 2000 tende a atribuir prioridades mais elevadas para *threads* do tipo *I/O bound*.

Em máquinas monoprocessadoras, a *thread* de mais alta prioridade está sempre ativa a menos que esteja bloqueada esperando por um evento (E/S ou sincronização). Caso exista mais de uma *thread* com um mesmo nível de prioridade o processador é compartilhado de forma *round-robin* entre essas *threads*. Em um sistema multiprocessador com *n* processadores, as *threads* de mais alta prioridade executam nos *n-1* processadores extras. As *threads* de mais baixa prioridade disputam o processador restante.

Na realidade, existe ainda um fator adicional que influencia fortemente o escalonamento do Windows 2000: a afinidade de uma *thread*. Uma *thread* pode definir sobre qual (ou quais) processador(es) ela deseja executar. Nesse caso, se a *thread* estiver apta a executar, porém o processador não estiver disponível, a *thread* é forçada a esperar, e uma outra *thread* é escalonada em seu lugar. O conceito de afinidade é motivado pela tentativa de reaproveitar os dados armazenados na cache do processador pela execução anterior de uma *thread*. O Windows 2000 possibilita dois tipos de afinidade: *soft* e *hard*. Por *default*, a política de afinidade *soft* é utilizada para escalonar *threads* a processadores. Nesse caso, o *dispatcher* tenta alocar uma *thread* ao mesmo processador que ela executou anteriormente, porém, se isso, não for possível, a *thread* poderá ser alocada a outro processador. Já com a política de afinidade *hard*, uma *thread* (com os devidos privilégios) executa em apenas um determinado processador.

10.5 Gerência de memória

O Windows 2000 implementa um sistema de memória virtual baseado em um espaço de endereçamento linear (plano) de 32 bits, o que fornece até 4 Gbytes de memória virtual. Esse espaço de endereçamento é normalmente dividido em duas partes de igual tamanho (2 Gbytes): uma destinada ao processo usuário (parte inferior) e outra parte destinada ao sistema operacional (parte superior). Em outros termos, uma aplicação (processo usuário) possui, no máximo, um tamanho de 2 Gbytes. Na realidade, o Windows 2000 oferece a opção de modificar essa alocação inicial de forma a atribuir 3 Gbytes ao processo usuário e 1 Gbyte ao sistema operacional. Essa possibilidade permite que certas aplicações, como, por exemplo, banco de dados, armazenem uma grande parcela de dados dentro do espaço de endereçamento da própria aplicação (processo). O Windows 2000 prevê ainda uma extensão, denominada de VLM (*Very Large Memory*) - destinada aos processadores de arquitetura de 64 bits - que permite a um processo usuário alocar até 28 Gbytes de memória virtual suplementar.

A alocação de memória por um processo Windows é realizada em duas fases. Inicialmente, o processo reserva um certo número de páginas da memória virtual sem necessariamente utilizá-las. Em seguida, à medida que o processo necessita de memória, essas páginas pré-alocadas são mapeadas a áreas efetivas de armazenamento no disco (área de *swap*). Essa segunda fase é conhecida, em terminologia Windows, como *commit*. Dessa forma, as páginas relativas ao espaço de endereçamento total de um processo usuário (2 Gbytes) podem estar em um de três estados: livres, reservadas ou dedicadas (*commited*). As páginas livres são as páginas do espaço de endereçamento não utilizadas pelo processo. As páginas reservadas correspondem àquelas pré-alocadas mas ainda não mapeadas a uma área de armazenamento real. Uma vez esse mapeamento

efetuado, as páginas pré-alocadas tornam-se páginas dedicadas. A distinção entre páginas reservadas e páginas dedicadas é justificada pela redução do tamanho do arquivo de paginação (*swap*) necessário a um processo. Apenas as páginas dedicadas consomem área de armazenamento. Por questões de desempenho, o Windows permite também que um processo, possuindo os privilégios necessários, bloqueie páginas em memória, fazendo com que essas páginas nunca sofram um procedimento de *swapping*.

Como a concepção do Windows é toda orientada a objetos, a memória alocada por um processo é representada através de um objeto memória. Dois processos podem compartilhar um mesmo espaço de endereçamento referenciando um objeto memória comum. Para o caso de compartilhamento de apenas uma região da memória o Windows oferece a abstração de visão (*view*). Essa abstração consiste em um processo mapear uma porção de seu espaço de endereçamento a um objeto (*section object*) o qual é utilizado por outros processos para acessos compartilhados a essa região. O mecanismo de *view* é bastante flexível. Ele permite que, em caso de *swapping* de uma região compartilhada de memória, as páginas correspondentes a essa região sejam transferidas ou para a área de *swap* ou para um arquivo especial (*mapped file*). É possível ainda fixar um endereço virtual para essa região compartilhada, permitindo assim que ela resida sempre em um mesmo endereço (virtual) em todos os processos que a utilizam. Finalmente, a área de *view* pode ter diferentes tipos de acesso, como, por exemplo, apenas leitura, leitura e escrita, execução, etc.

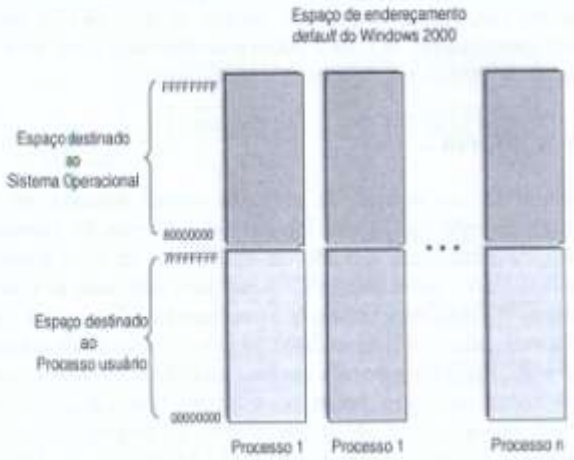

Figura 10.4 – *Layout* do espaço de endereçamento virtual do Windows 2000

10.5.1 Tradução de endereço virtual em endereço físico

A gerência de memória do Windows 2000 é baseada em paginação com um tamanho de páginas variando entre 4 Kbytes e 64 Kbytes, dependendo do processador. O mecanismo de tradução de um endereço virtual em um endereço físico é baseado em uma tabela de paginação em dois níveis. Nesse caso, considera-se que o endereço virtual de 32 bits é formado por três componentes: índice de diretório de páginas, índice da tabela de páginas, e índice de byte (deslocamento dentro da página).

A tradução de um endereço virtual em um endereço físico é realizada da seguinte forma. Inicialmente, utilizando a parte mais significativa do endereço virtual (índice de diretório de páginas), o diretório de páginas é acessado para determinar qual tabela de páginas está associada ao endereço virtual que se deseja traduzir. Uma vez definida a tabela de páginas, o índice de tabela de páginas é empregado para determinar a página correspondente a esse endereço virtual. A entrada da tabela de páginas fornece informações de controle e a localização em memória da página a ser acessada (se presente). Finalmente, o índice de byte é somado ao endereço inicial da página em memória, resultando no endereço físico correspondente ao endereço virtual desejado. A Figura 10.5 apresenta a relação entre esses três valores e a forma pela qual eles são utilizados para mapear um endereço virtual em endereço físico.

Cada processo possui um único diretório de páginas para mapear a localização das tabelas de páginas pertencentes a esse processo. O diretório de páginas possui 1024 entradas, o que limita o número máximo de tabelas de páginas de um processo. As tabelas de páginas são criadas sob demanda; por conseqüência, normalmente, muitas das entradas do diretório de página não são válidas. Cada tabela de páginas possui, por sua vez, também 1024 entradas, o que, de forma análoga, limita a quantidade de páginas por tabela de páginas. O tamanho de cada entrada, tanto do diretório de páginas como da tabela de páginas, é de 4 bytes. Isso implica que cada uma dessas estruturas ocupe 4 Kbytes de memória, ou seja, exatamente uma página (lembre-se de que o tamanho mínimo de páginas é 4 Kbytes). Essas tabelas são armazenadas no espaço de endereçamento virtual do processo, na área destinada ao sistema operacional (2 Gbytes superiores), e sua localização exata é mantida no descritor de processo.

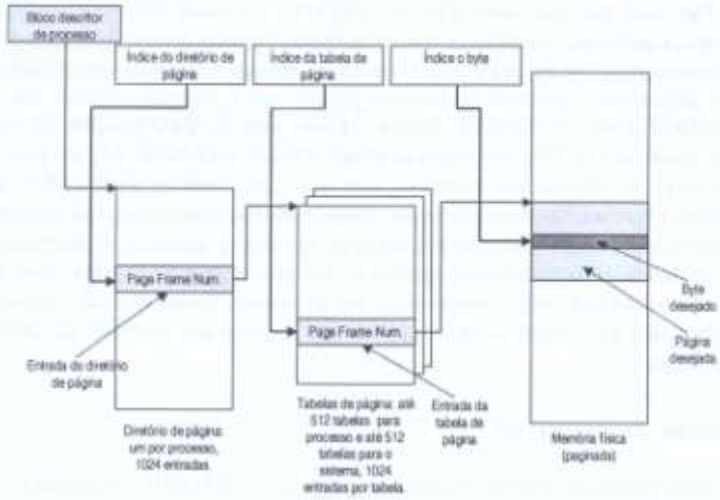

Figura 10.5 – Esquema de tradução de endereço virtual para endereço físico em arquiteturas Intel

10.5.2 Estratégias de paginação

Um sistema de gerência de memória baseado em paginação necessita determinar quando e como buscar páginas do disco (arquivo de paginação ou *swap*) para a memória. Além disso, é preciso definir, em caso de falta de espaço de memória, uma página a ser substituída para satisfazer a necessidade de carga de uma nova página. O algoritmo de paginação do Windows é baseado por demanda com *clustering* (em grupos). Nesse esquema, quando ocorre uma falta de página, o gerenciador de memória carrega na memória a página que faltava e mais um pequeno número de páginas ao seu redor. Essa estratégia tenta minimizar o número de acessos ao disco provocado pela paginação de um processo, explorando o princípio de localidade (um processo tende a executar, a todo instante, uma região limitada de seu espaço de endereçamento). A carga de páginas em avanço reduz o número de leituras individualizadas ao disco e, por conseqüência, o tempo de entrada e saída. A quantidade de páginas lidas em avanço (*cluster*) difere para páginas de código e páginas de dados e ainda varia conforme o tamanho da memória física.

A política para substituição de páginas na memória empregada pelo Windows depende do tipo do arquitetura da máquina e de seu processador. Para arquiteturas do tipo multiprocessador baseadas em processadores da família Intel, e em todas as máquinas baseadas em processadores da família Alpha, a estratégia utilizada é essencialmente FIFO local. Nessa estratégia, considera-se para fins de seleção da página a ser substituída em memória, apenas as páginas pertencentes ao processo. Nas arquiteturas de monoprocessadores Intel, o algoritmo de seleção de página a ser substituída é LRU, implementado através do algoritmo do relógio (*clock*).

O número de páginas presentes em memória para um processo é mantido através do mecanismo de *working set*. Para cada processo, esse número varia entre um valor mínimo e um valor máximo definidos apenas a partir do tamanho da memória física. Quando ocorre uma falta de página, os limites do tamanho do *working set* e a quantidade de memória livre são examinados. Havendo memória livre disponível, a gerência de memória permite que o processo aumente seu *working set* até atingir o valor máximo (na realidade, o valor máximo pode vir a ser ultrapassado em função da quantidade de memória livre). Se um processo atingir o limite máximo de seu *working set*, e ainda necessitar da carga de páginas em memória, sem que haja memória disponível, a gerência de memória iniciará o mecanismo de substituição, considerando apenas as páginas do *working set* do processo. Caso o número de faltas de página seja elevado, a gerência de memória verifica o *working set* de todos os processos em memória. Os processos que tiverem mais páginas em memória que o valor mínimo terão seus *working sets* reduzidos, e suas áreas de memória liberadas serão disponibilizadas para alocação global, isto é, um processo que necessite de memória poderá requisitar esse espaço.

10.6 Sistemas de arquivos

O Windows 2000 possui um sistema de arquivos próprio, o NTFS (NT *File System*), projetado de forma a oferecer segurança de acesso, garantia da consistência de dados em presença de falhas e suporte a discos de grande capacidade. O Windows 2000 oferece ainda suporte a outros sistemas de arquivos como o FAT (MS-DOS e Windows 3.1), FAT32 (Windows 95, 98, Millenium), o HPFS (OS/2), além de formatos para cdrom (CDFS) e UDF (*Universal Disk Format*) para acesso a dados armazenados em DVDs. O grande diferencial do NTFS em relação aos seus predecessores da linha Microsoft está em cobrir as necessidades de alguns pontos considerados como críticos para aplicações em ambientes corporativos, a saber:

- **Facilidade de recuperação de dados e tolerância a falhas:** para alcançar a confiabilidade, que é requisito nas aplicações corporativas, uma das técnicas utilizada pelo Windows 2000 é o processamento de transações. Uma transação é definida como uma operação de E/S que altera os dados do sistema de arquivo, ou a estrutura de diretório do volume, de forma indivisível, isto é, cada alteração no sistema de arquivo só é considerada efetivada se for completamente realizada. O NTFS emprega esse modelo de transações para implementar seu recurso de recuperação do sistema de arquivos permitindo que este seja reconstruído, ou, ao menos, mantido em um estado consistente, após uma falha no sistema. Além disso, o Windows 2000 oferece suporte para RAID.

- **Segurança:** o NTFS explora o modelo de objetos para oferecer segurança aos arquivos. Um arquivo aberto é implementado como um objeto arquivo o qual atua como um descritor, definindo os diferentes privilégios de acesso e requisitos de segurança a este.

- **Suporte a grandes discos e arquivos:** o NTFS é projetado para suportar de forma eficiente acesso, manipulação e armazenamento em discos de grandes capacidades.

- **Fluxos de dados múltiplos:** um arquivo e seus atributos são vistos como uma seqüência de bytes, denominada fluxo de dados. Dessa forma, no NTFS, um único arquivo pode ter associado vários fluxos de dados. Essa organização oferece uma grande flexibilidade pois permite que o fluxo de dados que compõe o arquivo seja interpretado de acordo com o fluxo de dados de seu atributo.

- **Facilidades de indexação:** o NTFS permite que arquivos sejam acessados de forma indexada através de atributos (chaves de pesquisa) criados para cada arquivo.

- **Suporte para sistema POSIX:** o NTFS implementa os recursos exigidos pelo POSIX como diferenciação de maiúsculas e minúsculas para nomes de arquivos e diretórios e atalhos (*soft links*).

O NTFS é organizado sobre três estruturas básicas: setor, *cluster* e volume. O setor, normalmente composto de 512 bytes, é a menor unidade de alocação física do disco. Setores contíguos podem ser organizados em grupos, formando os *clusters*. Um *cluster* constitui então a unidade básica de alocação do NTFS, isto é, um arquivo ocupa em disco sempre um número de bytes múltiplo do tamanho do *cluster*. Um volume corresponde a uma partição lógica do disco. Um volume é composto por uma série de *clusters* e possui de forma autocontida informações relacionadas a esse disco lógico, isto é, estrutura de diretório, *clusters* livres e ocupados, etc. Atualmente, o número máximo de *clusters* permitido a um arquivo no NTFS é 2^{32}, sendo o tamanho máximo de um *cluster* limitado a 64 kbytes, o que nos leva a arquivos de até, no máximo, 2^{48} bytes.

Um volume (disco lógico ou partição) é organizado em 4 regiões. A primeira região de um volume NTFS corresponde ao setor de boot, que na realidade pode ocupar até 16 setores físico do disco, apesar do nome "setor de boot". O setor de boot possui informações sobre o *layout* do volume, a estrutura do sistema de arquivos e o programa de boot do Windows. Essa região é seguida pela *Master File Table* (MFT) a qual contém as informações sobre todos os arquivos e diretórios (*folders*) desse volume, assim como sobre o espaço livre. O MTF é organizado na forma de um conjunto de registros de tamanho variável, em que cada arquivo ou diretório possui um registro associado. Nesses registros, está incluído o próprio MTF, já que ele também não deixa de ser um tipo de arquivo. Cada registro do MTF mantém informações relacionadas ao atributo do arquivo

(leitura, escrita, etc.), datas de criação e modificação, nome do arquivo, descritor de segurança. A terceira região é a dos arquivos de sistema, tipicamente de 1 Mbyte, em que são armazenados: uma cópia parcial da MFT (informações suficientes para recuperar erros físicos acontecidos na MFT); um arquivo de *logs*, referente ao controle de transações do NTFS; um *bit map* que fornece a ocupação dos *clusters* do volume; e uma tabela de atributos que define o tipo de acesso a esse volume (seqüencial, indexado, etc). Finalmente, a quarta e última região corresponde à área disponível para os arquivos.

Como nós mencionamos anteriormente, um dos principais objetivos do NTFS é facilitar a recuperação em caso de falhas. A capacidade de recuperação do NTFS é essencialmente baseada em *logs* de transação. Uma operação que altere o sistema de arquivos é tratada como uma transação, a qual é gravada na região de *logs* associada a cada volume. Apenas após a gravação da transação é que a operação é efetivada. É importante salientar que a capacidade de recuperação do NTFS foi projetada para garantir a coerência e a recuperação de estruturas e dados do sistema operacional e não dados de arquivos de usuário. Dessa forma, um usuário jamais perderá o acesso a um volume (partição) em decorrência de uma falha física, ou do sistema, embora possa perder acesso ao conteúdo de um arquivo seu. Entretanto, para suprir a necessidade de recuperação total, o Windows disponibiliza uma série de ferramentas para a tolerância a falhas; entre elas, podemos citar o suporte a RAID.

10.7 Gerência de entrada e saída

O sistema de gerência de entrada e saída (E/S) do Windows 2000 é responsável pelos acessos ao sistema de arquivos, pelo gerenciamento da cache de dados do sistema operacional, pelos drivers de dispositivos e pelo driver de rede. O fato de o driver de rede ser visto de forma separada dos demais drivers de dispositivos é conseqüência do objetivo do Windows 2000 oferecer um suporte a ambientes distribuídos. O driver de rede realiza muito mais que o simples envio e recepção de pacotes de dados são de sua responsabilidade todas as facilidades de rede de o Windows 2000, o que justifica esse tipo de "tratamento especial".

As requisições de E/S são todas convertidas pela gerência de E/S do Windows 2000 em um formato padrão denominado de IRP (*I/O Request Packet*). Em seguida, o IRP é direcionado ao driver de dispositivo responsável pela operação a ser realizada. Quando a operação é finalizada, o driver de dispositivo sinaliza a gerência de E/S. Esse mecanismo oferece suporte para operações tanto assíncronas como síncronas.

10.7.1 A interface WDM

Plug-and-play é a capacidade que um sistema operacional possui de reconhecer e adaptar-se de forma dinâmica a alterações na sua configuração inicial de hardware. Através do *plug-and-play* um usuário pode inserir e remover periféricos ao seu sistema sem se preocupar com sua configuração, ou interferência com os demais componentes do sistema. A Microsoft segue a filosofia *plug-and-play* desde o sistema operacional Windows 95, porém esse mecanismo sofreu muitas evoluções. A mais importante evolução do mecanismo de *plug-and-play* foi a definição, e posteriormente a adoção, da especificação ACPI (*Advanced Configuration and Power Interface*) por parte de vários fabricantes de placas-mãe (*motherboard*). Basicamente essa especificação retira da BIOS o controle do *plug-and-play* e a gerência de energia para deixá-los a cargo do sistema operacional. As funcionalidades previstas pela especificação de ACPI são independentes de sistema operacional e de processador e possuem uma interface bem definida. Essa interface é utilizada pelos projetistas de sistemas operacionais na concepção de uma série de serviços para a gerência de entrada e saída.

No caso específico do Windows 2000, as melhorias introduzidas na funcionalidade de *plug-and-play* visam a simplificar o desenvolvimento de drivers de dispositivos e de sua gerência. Na prática, isso se traduz na unificação dos diferentes módulos "assistentes de instalação" (*wizards*) existentes para diferentes tipos de periféricos em um único "assistente de instalação". Essa abordagem baseia-se no emprego de um modelo genérico para implementação e gerência de drivers de dispositivos, o WDM (*Win32 Driver Model*). O modelo WDM é um padrão a ser seguido no desenvolvimeno de drivers de dispositivos de forma a permitir a fácil portabilidade desses de um sistema operacional a outro. Os drivers de dispositivos que seguem o WDM possibilitam uma compatibilidade a nível de código binário entre todas as plataformas baseadas em processadores x86 executando Windows 2000 e Windows 98, e são portáveis a nível do código fonte para qualquer outra arquitetura.

10.7.2 O suporte a RAID

O Windows 2000 oferece suporte a RAID por hardware e por software. O suporte a RAID por hardware na realidade significa que o Windows 2000 oferece drivers de disco com capacidade de gerenciar controladoras RAID. Assim, discos físicos distintos podem ser combinados de diferentes maneiras para compor um ou mais discos lógicos. No RAID por hardware, é a própria controladora que gerencia a criação e a manutenção da informação de redundância necessária à recuperação de dados.

O suporte a RAID via software, disponível apenas nas versões *server* do Windows 2000, emula, a partir de serviços do próprio sistema operacional, o funcionamento de uma controladora com suporte RAID. Um driver de dispositivo (FTDISK) é responsável por essa tarefa. Esse driver oferece RAID 1 e RAID 5.

10.8 O serviço de Active Directory

Uma das principais novidades introduzidas no Windows 2000 é a inclusão do serviço de diretório denominado de *active directory*. Mas o que é um serviço de diretório? No contexto de uma rede, um diretório é uma estrutura hierárquica que armazena informações a respeito de itens (recursos) de uma rede. Um diretório pode ser composto pelos mais diferentes tipos de itens (objetos), como por exemplo, servidores, discos, impressoras, contas de usuários, arquivos compartilhados; ou ainda domínios de *logon*, aplicativos, políticas de segurança e de acesso, etc. Cada item é um objeto e possui uma série de informações (atributos) associados a ele. Por exemplo, os atributos de um determinado objeto usuário podem ser nome, endereço, nome de usuário (*login name*), senha de acesso e grupos de que faz parte. O conceito de serviço de diretório é o conjunto de funções para criação, armazenamento e recuperação de informações de um diretório. O *active directory* do Windows 2000 tem exatamente esse objetivo, isto é, permitir que objetos sejam criados e manipulados facilmente.

O *active diretory* implementa um espaço de nomes. Um espaço de nomes é uma área em que um determinado nome pode ser resolvido, isto é, transformado em um objeto ou nas informação que representa. Esse processo de transformação é denominado de resolução de nomes. Um exemplo de um espaço de nomes e de resolução é o guia telefônico. Nesse caso, os nomes de assinantes são resolvidos para seus respectivos números de telefones. A estrutura do *active directory* é fortemente baseada em outro exemplo de espaço de nomes bastante comum no dia-a-dia de ambientes de redes: o DNS (*domain name server*). Devido a essa forte relação, nós iremos, de uma forma bastante sucinta, apresentar o princípio de funcionamento do DNS.

O DNS é uma hierarquia de domínios que representa toda a Internet em um único espaço de nomes. Essa hierarquia é organizada na forma de uma árvore. O nível logo abaixo da raiz corresponde aos domínios de mais alto nível (*top level domain - TLD*), os quais são subdivididos em domínios de segundo nível, e estes por sua vez em domínios de terceiro nível e assim sucessivamente. Os domínios TLD correspondem aos domínios associados a grandes categorias, como por exemplo, comercial (*.com*), governamental (*.gov*), educacional (*.edu*), ou ainda a países, Brasil (*.br*), França (*.fr*), etc. Esses domínios são administrados pela InterNIC (*Internet Network Information Center*). A InterNIC delega a responsabilidade de gerência do segundo nível a entidades administrativas oficialmente reconhecidas e registradas. No caso do domínio *.br* (Brasil) essa entidade é a FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo). A FAPESP, por sua vez, administra domínios de segundo nível como *.com.br* (comercial), *.gov.br* (governamental), ou ainda domínios do tipo *.ufrgs.br*. A resolução de nomes de um domínios DNS consiste em, dado o nome de uma máquina em um domínio, obter o endereço IP equivalente. A resolução de nomes do DNS é baseada em sua estrutura hierárquica. Assim, um máquina identificada como, por exemplo, *asterix.inf.ufrgs.br* diz respeito a um computador (*asterix*) que existe no domínio *inf* que pertence ao domínio *ufrgs* no Brasil (*.br*). É responsabilidade do domínio *inf.ufrgs.br* resolver o nome da máquina *asterix*, isto é, retornar qual é seu endereço IP.

O *active directory* utiliza a mesma estrutura de nomes dos domínios DNS. Entretanto, é importante observar que, apesar de empregar nomes idênticos, eles não compõem um único espaço de nomes, ou seja, em cada espaço de nomes (DNS ou *active directory*), os nomes são resolvidos para informações diferentes. Na realidade, o DNS é um serviço de resolução puro, isto é, um cliente DNS envia uma requisição a um servidor DNS para "traduzir" um nome de máquina em IP. Já o *active directory* é um serviço de diretório. A resolução do nome depende do tipo de objeto que o nome representa. Nesse procedimento, um cliente *active directory* realiza requisições a um servidor *active directory* (também denominado de controlador de domínio) através de um protocolo específico: o LDAP (*Lightweight Directory Access Protocol*). O LDAP é um protocolo desenvolvido dentro da Internet para fornecer acesso a serviços de diretório. O fato de o *Windows* empregá-lo oferece a este um grau adicional de conectividade a ambientes heterogêneos.

Mas, de uma forma pragmática, o que finalmente é, e para que serve o *active directory* ? O objetivo principal do *active directory* é facilitar a adminstração da rede. Versões anteriores do Windows incluíam uma série de serviços e conceitos para auxiliar usuários e administradores de redes a localizar e gerenciar recursos da rede. O *network neighborhood*, o *WINS Manager*, o *Server Manager* são alguns exemplos desses serviços. Os objetos gerenciados por essas ferramentas consistiam no que se denominava de domínio Microsoft Windows NT. Essa abordagem, entretanto, mostrou-se um pouco prática para usuários e administradores – especialmente quando a rede apresentava uma certa complexidade pois forçava a sua subdivisão em vários domínios Windows NT, o que dificultava a localização e o gerenciamento de seus recursos. O *active directory* foi concebido para executar esse mesmo serviço, porém de uma maneira mais eficiente.

Os principais componentes que formam o *active directory* são: objeto, esquema, contêiner. Um objeto é qualquer usuário, sistema, recurso ou serviço existente dentro do *active directory*. Os objetos são descritos por seus atributos, como por exemplo, nome de uma máquina e seu endereço IP. O conjunto de atributos para qualquer tipo particular de objeto é chamado de esquema. Um contêiner é um tipo especial de objeto utilizado para organizar o *active directory*. Sua idéia é similar de *folder* (pasta) do *Windows*, isto é, se um *folder* contém arquivos e outros *folders*, um contêiner armazena objetos e outros contêiners. Os três tipos possíveis de contêiner são domínios, sites e unidades organizacionais.

Um domínio em Windows 2000 é muito similar ao conceito de domínio do Windows NT 4.0. Um domínio é um grupo de usuários e computadores que formam uma unidade administrativa isolada. Um site consiste em uma localização geográfica empregada para distinguir localizações remotas de localizações locais. Os sites podem ser comparados a subredes, isto é, sua estrutura pode ser empregada por aplicativos para localizar um determinado servidor mais próximo a esta subrede, reduzindo assim o tráfego em redes remotas. Uma unidade organizacional é um contêiner utilizado para agrupar objetos com políticas de acesso idênticas, podendo ser criada com base em vários critérios, como função, localização, recursos, etc. As unidades organizacionais existem dentro de um domínio.

Uma característica das grandes organizações é a necessidade de criar vários domínios para controlar de forma mais apropriada os recursos de um determinado setor, departamento, filial, etc. O *active directory* permite que os domínios sejam organizados hierarquicamente na forma de uma árvore. Essa organização cria uma relação de filiação entre os domínios com uma relação de confiança: um pai confia em seu filho, e o filho confia em seu pai. Essa relação de confiança permite que recursos sejam compartilhados entre pais e filhos, e entre filhos de um mesmo pai. Cada árvore possui um espaço de nome próprio. O conjunto de árvores, isto é, de espaços de nomes diferentes define o que se denomina, em terminologia Windows, de uma floresta.

O *active directory* é disponível no Windows 2000 apenas na sua versão server e, além de prover meios para o armazenamento de dados e acesso a serviços de um diretóiro, ele também integra mecanismos de segurança para evitar acessos não autorizados a objetos e mecanismos de replicação para garantir um certo grau de tolerância a falhas.

10.9 O serviço de *cluster*

Um *cluster* é conceituado como sendo uma coleção de sistemas independentes conectados entre si para executar um conjunto de aplicações, fornecendo a "ilusão" de ser um único sistema. A principal motivação de incluir no Windows 2000 capacidade de *clustering* foi o fato de se desejar garantir o funcionamento ininterrupto de serviços em ambientes corporativos. O suporte a *clustering* existe apenas nas versões *Advanced Server* e *Datacenter Server*.

O Windows 2000 oferece duas tecnologias de *clustering* que podem ser utilizadas de forma individual ou combinada. A primeira, serviço de *cluster*, tem como principal objetivo melhorar o tempo de funcionamento do sistema através da recuperação de falhas (mecanismo de *fail-over*). A idéia básica consiste em possibilitar que servidores sirvam de *backup* um do outro. Nessa configuração, quando um servidor principal falha, um segundo servidor assume automaticamente a responsabilidade da realização do serviço. Essa facilidade pode ainda ser explorada para atividades de manutenção agendadas. Nesse caso, é possível que um servidor *backup* continue a prover o serviço enquanto o servidor principal está sendo reparado. A segunda tecnologia de *clustering* disponível no Windows 2000 tem por objetivo melhorar a escalabilidade do sistema mediante o equilíbrio de carga entre múltiplas máquinas. Esse serviço é denominado de NLB (*Network Load Balancing*) e permite que o tráfego IP associado a um tipo de requisição (serviço) seja dividido em até 32 máquinas diferentes, configuradas para realizar um determinado serviço.

A utilização eficiente de *clusters* depende primariamente de três fatores: facilidade de implementação (configuração) do *cluster*, desenvolvimento de programas e gerenciamento. O Windows 2000, nas versões *Advanced Server* e *Datacenter Server*, incluem um "wizard" para auxiliar a criação do *cluster* e para facilitar a sua integração com serviços e aplicações distribuídas baseadas em DCOM. Os serviços de *clustering* no Windows 2000 estão embutidos no próprio