5

Sistemas Operacionais como Programas Concorrentes

Rômulo Silva de Oliveira *(UFSC, romulo@das.ufsc.br)*[1]
Alexandre da Silva Carissimi *(UFRGS, asc@inf.ufrgs.br)*[2]
Simão Sirineo Toscani *(UFRGS, PUCRS, simao@inf.ufrgs.br)*[3]

Resumo:

Este texto discute as possíveis organizações para um *kernel* (núcleo) de sistema operacional. Inicialmente é feita uma rápida revisão dos conceitos fundamentais dos sistemas operacionais e dos mecanismos clássicos para a sincronização de processos. Em seguida, diversas organizações possíveis para o *kernel* do sistema operacional são apresentadas. Em muitas delas o *kernel* aparece como um programa concorrente, onde a preocupação com a sincronização entre processos está sempre presente. No final, o *kernel* do Linux é descrito, como um estudo de caso.

[1] Doutor em Engenharia Elétrica, ênfase em Sistemas de Informação, pela UFSC e Mestre em Ciência da Computação pelo Programa de Pós-Graduação em Computação da UFRGS. Professor do Instituto de Informática da UFRGS de 1989 a 1999, atualmente é Professor do Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina e orientador de Mestrado junto ao Programa de Pós-Graduação em Computação da UFRGS. DAS-CTC-UFSC, Caixa Postal 476, Florianópolis-SC, 88040-900

[2] Doutor em Informática pelo Instituto Nacional Politécnico de Grenoble (INPG), Franca. Mestre em Ciência da Computação pelo Programa de Pós-Graduação em Computacao e Engenheiro Eletricista pela UFRGS. Professor do Departamento de Informatica Aplicada da UFRGS. Caixa Postal 15064, Porto Alegre - RS, 91501-970

[3] Doutor em Informática pela Universidade Nova de Lisboa, Portugal. Mestre em Informática pela PUC/RJ. Engenheiro eletricista pela UFRGS. Professor do Instituto de Informática da UFRGS de 1975 a 1998. Atualmente, professor da Faculdade de Informática, PUCRS e orientador de Mestrado junto ao

Programa de Pós-Graduação em Computação da UFRGS. Av. Ipiranga, 6681 - Prédio 30, bloco 4, Porto Alegre – RS, 90619-900

4.1. Introdução

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 fig. 5.1. Esta seção contém uma revisão dos conceitos fundamentais dos sistemas operacionais e está baseada em [OLI 01].

Figura 5.1: Sistema computacional.

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. Tipicamente isso é obtido através do compartilhamento de recursos entre programas, como, por exemplo, espaço na memória principal, tempo de processador, impressora, espaço e acesso a disco. Uma utilização mais conveniente implica na diminuição no tempo necessário para a construção dos programas e aprendizado para utilização do Sistema. Isso é obtido, escondendo-se do programador detalhes do hardware e do sistema operacional. Isso também implica a redução no custo do software, pois são necessárias menos horas de programador.

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. 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.

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 um computador, passa a ser importante gerenciar os recursos que cada usuário emprega. Essa informação pode ser utilizada tanto para fins de cálculo de um valor a ser cobrado pelo uso do computador como na identificação de gargalos dentro do sistema. O compartilhamento de um computador, ou recursos em geral, só 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. Cabe ao sistema operacional garantir que cada usuário possa trabalhar sem sofrer interferência danosa dos demais.

Os programas solicitam serviços ao sistema operacional através das **chamadas de sistema**, também conhecidas como API (*Application Program Interface*). Elas são semelhantes às chamadas de subrotinas. Entretanto, enquanto as chamadas de subrotinas são transferências para procedimentos normais do programa, as chamadas de sistema transferem a execução para o código do sistema operacional. O retorno da chamada de sistema, assim como o retorno de uma subrotina, 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 subrotinas de uma biblioteca. São as subrotinas 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. 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.

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.

O mais importante programa de sistema é o **interpretador de comandos (*shell*)**, o qual é ativado pelo sistema operacional sempre que um usuário inicia sua sessão de trabalho. Sua tarefa é receber comandos do usuário e disparar sua execução. Algumas vezes 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.

4.1.1. Multiprogramação

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

forma, processador e periférico trabalham ao mesmo tempo. Enquanto o processador executa o programa 2, o periférico realiza a operação solicitada pelo programa 1.

Em sistemas operacionais é conveniente diferenciar um programa de sua execução, para tanto é usado o conceito de processo. Não existe uma definição objetiva, aceita por todos, para a idéia de processo. Na maioria das vezes, um **processo** é definido como "um programa em execução". O conceito de processo é 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.

A descrição acima mostrou diversos momentos pelos quais passa o processo durante a sua existência. A partir dessa descrição, pode-se estabelecer os **estados** possíveis para um processo. Após ser criado, o processo 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 operação de E/S do processo 1, ele precisa de processador para voltar a executar. Como o mesmo 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. Suponha 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*).

No **estado executando**, um processo efetivamente tem seu código executado pelo processador e pode fazer chamadas de sistema. Até a chamada de sistema ser atendida, o processo não pode continuar sua execução. Ele fica bloqueado e só volta a disputar o processador após a conclusão da chamada. Enquanto espera pelo término da chamada de sistema, o processo está no **estado bloqueado** (*blocked*).

A mudança de estado de qualquer processo é iniciada por um evento. Esse evento aciona o sistema operacional, que então altera o estado de um ou mais processos. Como visto antes, a **transição do estado** de 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 da mesma. 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*). Existem soluções de escalonamento onde o processo executando somente libera o processador voluntariamente, ao solicitar uma operação que leve ao seu bloqueio. Este tipo de solução é dita **não-preemptiva**. Quando o escalonamento, por ser baseado em prioridades ou fatias de tempo, pode transferir automaticamente um processo do estado executando para o estado apto, então, tem-se uma solução do tipo **preemptiva**.

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 fig. 5.2 mostra o grafo de estados dos processos com esses novos caminhos.

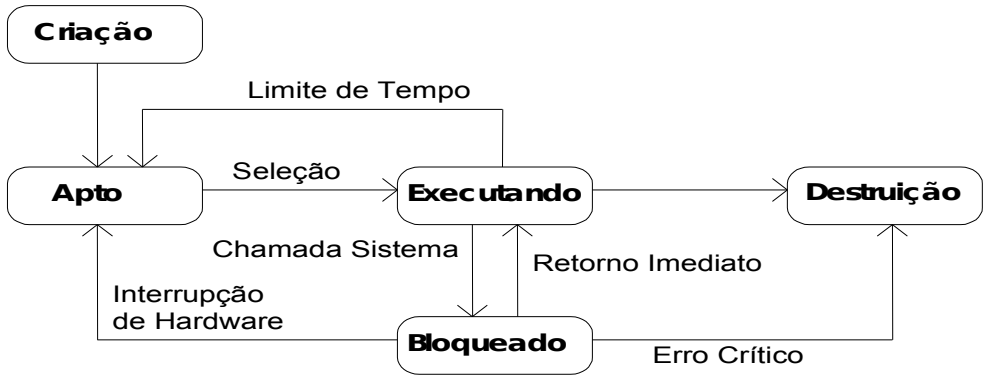

Figura 5.2: Diagrama de estados de um processo.

4.1.2. Mecanismo de interrupções

Talvez o mais importante conceito na realização de um sistema operacional seja o conceito de interrupção. É através de seu emprego que a multiprogramação, chamadas de sistema, e os mecanismos de gerência de memória, gerência de E/S, entre outros, são implementados.

O **mecanismo de interrupção** é um recurso comum a todos os processadores empregado para sinalizar a ocorrência de um evento e solicitar a intervenção do processador para que esse evento seja tratado de forma adeqüada. Fisicamente, um barramento de controle é usado para enviar sinais elétricos associados a ocorrência evento. Essa é forma empregada para que o evento "chame a atenção" do processador. Em resposta a essa ação, o processador desvia (interrompe) a posição atual do programa em execução para uma rotina específica responsável por realizar uma série de ações relacionadas ao tratamento dessa interrupção. A rotina responsável por atender a

interrupção é chamada de **tratador de interrupção**. Quando o tratador termina, a execução retorna ao ponto em que o programa se encontrava sem que esse perceba que foi interrompido.

Em certos aspectos, uma interrupção é semelhante a uma chamada de subrotina: em ambos casos existe uma rotina que é ativada e, quando termina, a execução retorna ao ponto original. Entretanto, interrupções diferem drasticamente na forma pela qual são ativadas. Por serem vinculadas a ocorrência de eventos externos, o momento exato de sua ativação não pode ser previsto pelo programa podendo se situar em qualquer ponto de um programa (interrupções de software, como será visto mais adiante, são exceções). Entretanto, existem momentos em que um programa não pode ser interrompido, como, por exemplo, quando está alterando variáveis que também são acessadas pelo tratador de interrupção. Se uma interrupção ocorrer nesse instante, o tratador será ativado e irá acessar variáveis que podem estar com seus valores temporariamente inconsistentes por estarem sendo manipuladas pelo programa. É necessário então impedir que isso ocorra.

A solução para esse problema é desligar temporariamente o atendimento a interrupções, enquanto o programa realiza uma tarefa crítica que não pode ser interrompida. Os processadores normalmente dispõem de instruções **para habilitar e desabilitar interrupções**. Enquanto as interrupções estiverem desabilitadas, o tratador de interrupção não é acionado.

Ainda, para que a execução do programa interrompido não seja comprometido é necessário que a execução do tratador de interrupção mantenha o estado do processador inalterado. Em outras palavras, o conteúdo de todos registradores internos deverá ser o mesmo que no momento que ocorreu a interrupção. Alguns processadores salvam automaticamente o contéudo de todos registradores quando ocorre uma interrupção. Outros processadores salvam apenas alguns, cabendo à rotina que atende à interrupção salvar os demais registradores. O tratador de interrupção termina ao executar uma instrução especial de **retorno da interrupção**. Essa instrução restaura o contéudo original dos registradores, salvos no momento da ativação, e faz o processador retomar a execução do programa interrompido.

Uma questão ainda persiste. Como em um computador podem ocorrer várias interrupções é necessário identificar a fonte da interrupção para realizar o procedimento apropriado. A maioria dos processadores admitem diversos tipos de interrupções. Cada tipo de interrupção é identificado por um número, como, por exemplo, de zero a 255. Nesse caso, a solução mais simples consiste em atribuir um tipo diferente a cada interrupção. Nesse caso, a ocorrência de um evento externo não somente interrompe o processador, mas também a identifica através de seu tipo. O endereço de um tratador de interrupção é chamado 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. Ela é chamada 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 interrupção 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 subrotina, pois o próprio programa interrompido é quem gera a interrupção, levando à execução do tratador correspondente. A vantagem sobre subrotinas 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.

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 ou não foi previamente alocada para o processo. O procedimento de atendimento a essa classe de interrupções é igual ao descrito anteriormente.

4.1.3. Mecanismos de proteção

Na multiprogramação diversos processos compartilham o computador. É necessário que o sistema operacional ofereça proteção aos processos e garanta a utilização correta do 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 e o sistema operacional realiza todas as inicializações necessárias.

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. A execução de uma **interrupção de software** ou ***trap*** implica também em chavear o processador para modo supervisor (lembre que as chamadas de sistemas são normalmente empregadas para implementar as chamadas de sistema).

A fig. 5.3 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.

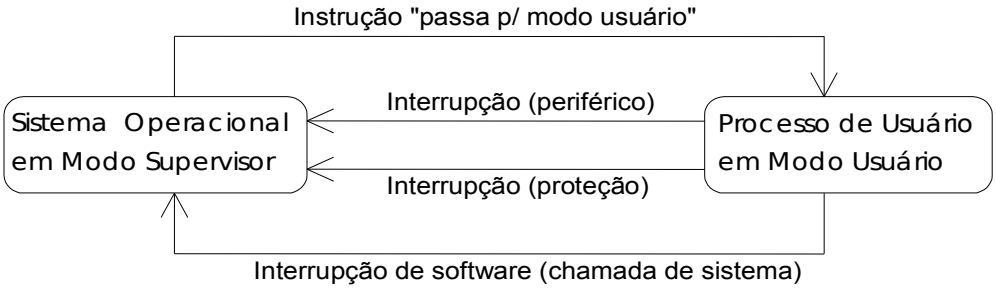

Figura 5.3: Modos de operação do processador.

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.

Para implementar a proteção de memória, o sistema operacional também necessita de auxílio da arquitetura do hardware. Existem muitas maneiras de realizar a proteção da memória, sendo a mais importante apresentada mais adiante (seção 1.1.5).

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 suspender o processo por exceder o limite de tempo de execução. Mais tarde, o processo suspenso terá 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 garante 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.

4.1.4. Chaveamento de processos

A base da multiprogramação é o compartilhamento do processador entre os processos. Por exemplo, enquanto o processo 1 fica bloqueado, à espera de um dispositivo periférico, o processo 2 ocupa o processador. Em sistema multiprogramado, é necessário interromper processos para continuá-los mais tarde. Essa tarefa é chamada de **chaveamento de processo**. 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** inclui o conteúdo dos 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. 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 registrador 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*.

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*. 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 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. 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 problemas de 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.

4.1.5. 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 acessar. Os endereços manipulados pelo processo são endereços lógicos. Em outras palavras, as instruções de máquina de um processo especificam endereços lógicos. Por exemplo, um processo executando um programa escrito na linguagem C manipula variáveis tipo *pointer*. Essas variáveis contêm endereços lógicos. Em geral, cada processo possui a sua memória lógica, que é independente da memória lógica dos outros processos.

A **memória física** é aquela implementada pelos circuitos integrados de memória, pela eletrônica do computador. O endereço físico é aquele que vai para a memória física, ou seja, é usado para endereçar os circuitos integrados de memória.

O **espaço de endereçamento lógico** de um processo é formado por todos os endereços lógicos que esse processo pode gerar. Existe um espaço de endereçamento lógico por processo. Já o **espaço de endereçamento físico** é formado por todos os endereços aceitos pelos circuitos integrados de memória. A **unidade de gerência de memória** (*Memory Management Unit*, MMU) é o componente do hardware responsável por prover os mecanismos que serão usados pelo sistema operacional para gerenciar a memória. Entre outras coisas, é a MMU que vai mapear os endereços lógicos gerados pelos processos nos correspondentes endereços físicos que serão enviados para a memória. A fig. 5.4 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.

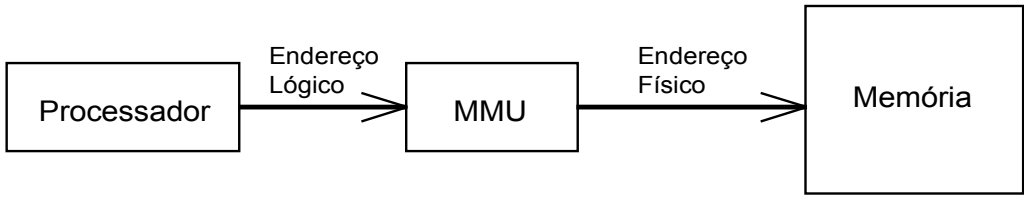

Figura 5.4: Diagrama incluindo a MMU entre o processador e a memória.

A fig. 5.5 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. Pode-se 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.

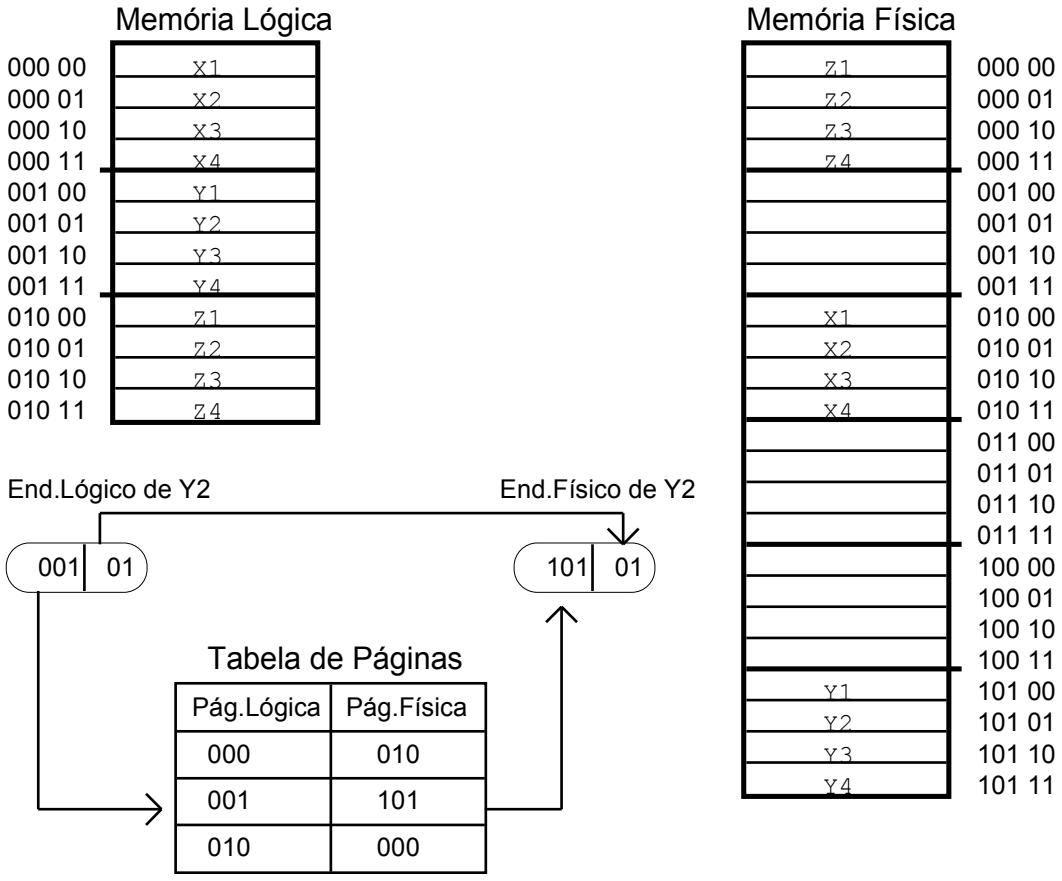

Figura 5.5: Mecanismo básico de paginação.

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 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. Durante a carga é montada uma **tabela de páginas** para o processo. Essa tabela informa, para cada página lógica, qual a página física correspondente.

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.

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 fig. 5.5 mostra como o endereço lógico do *byte* Y2 é transformado no endereço físico correspondente.

Um aspecto importante da paginação é a forma como a tabela de páginas é implementada. Observe que ela deve ser consultada a cada acesso à memória. Uma solução é manter a tabela de páginas na própria memória. 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. 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. Quando a entrada da tabela de páginas associada

com a página lógica acessada não está na TLB, 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. Quando a TLB 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 TLB 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.

4.2. Revisão de programação concorrente

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 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 texto 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 estabelecido no Brasil, sendo algumas vezes usado o termo **programação paralela**.

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 se vai tratar nessa seção. Maiores informações podem ser encontradas em [OLI 01].

4.2.1. Problema da seção crítica

Uma forma de implementar a 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 ou algumas variáveis.

No entanto, o compartilhamento de uma mesma região de memória por 2 ou mais processos pode causar problemas. 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.

Chama-se de **seção crítica** aquela parte do código de um processo que acessa uma estrutura de dados compartilhada. 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. Por exemplo, isso significa que, enquanto um processo estiver inserindo nomes em uma fila, outro processo 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:

- A solução não depende das velocidades relativas dos processos;
- 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;
- Existe exclusividade mútua entre os processos com referência a execução das respectivas seções críticas.

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. Isto pode ocorrer porque cada um detém um recurso que o outro precisa, e solicita o recurso que o outro detém. Como nenhum dos dois vai liberar o recurso que possui antes de obter o outro, ambos ficarão bloqueados indefinidamente.

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 por alguns sistemas operacionais.

Outra solução possível para o problema da seção crítica é o ***Spin-Lock*.** Essa solução é baseada em instruções de máquina que permitem trocar entre si o valor de uma posição de memória com o valor contido em um registrador. Essa troca é feita de forma indivisível, isto é, sem ser interrompida antes de seu término. As intruções "*Test-and-Set*" e "*Compare on Store*", disponíveis em processadores comerciais, são exemplos típicos.

Com base no comportamento dessas instruções é possível proteger a entrada da seção crítica 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 lock = 0;
```

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;
            Test_and_set( 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 prossegue 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.

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.

4.2.2. 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. Pode-se 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.valor <= 0
            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).

Uma variação muito comum de semáforos são as construções ***mutex*** ou **semáforo binário.** Nesse caso, tem-se 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:

```
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 */
```

4.2.3. Mensagens

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 parâmetro o identificador do processo do qual se deseja receber a mensagem. Nesse caso, a chamada somente será satisfeita com uma mensagem do processo especificado.

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.

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 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, tem-se os sistemas que não oferecem *buffers* para as mensagens e o remetente é sempre bloqueado até o respectivo destinatário executar Receive.

4.3. Organização de sistemas operacionais

Um sistema operacional também é um programa de computador e, como tal, possui uma especificação e um projeto. A especificação do mesmo corresponde à lista de serviços que deve executar e as chamadas de sistema que deve suportar. Por outro lado, o seu **projeto** ou ***design*** diz respeito à sua estrutura interna, como as diferentes rotinas necessárias na implementação dos serviços são organizadas internamente. O tamanho de um sistema operacional pode variar desde alguns milhares de linhas no caso de um pequeno núcleo para aplicações embutidas (*embedded*) até vários milhões de linhas, como na versão 2.4 do Linux, chegando a 30 milhões de linhas no caso do Windows 2000 [STA 01]. Embora princípios básicos como baixo acoplamento e alta coesão [PRE 01] sejam sempre desejáveis, existem algumas formas de **organização interna** para sistemas operacionais que tornaram-se clássicas. Também ao longo do tempo a terminologia sofreu variações. A forma como os termos são apresentados neste texto procura criar uma taxonomia coerente e didática, mesmo que alguns autores, em alguns momentos, possam ter usados os termos com um sentido ligeiramente diferente.

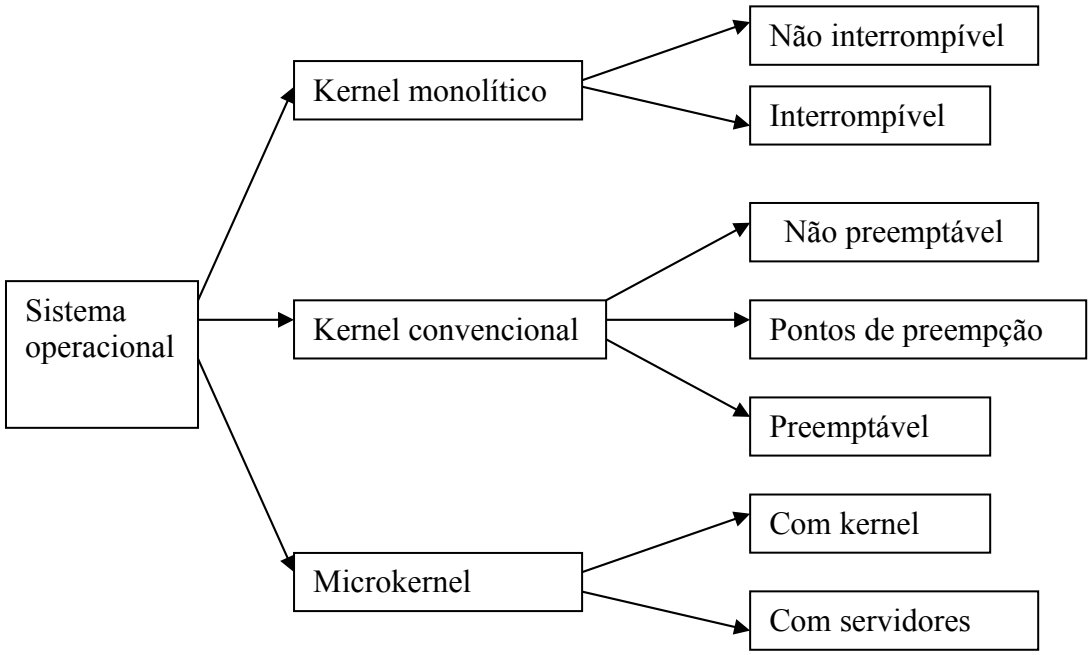

Figura 5.6: Taxonomia das organizações de sistemas operacionais.

Ao longo dos últimos 40 anos, sistemas operacionais têm crescido de tamanho e de complexidade. Conforme o histórico que aparece em [STA 01], o CTSS, criado pelo MIT em 1963, ocupava aproximadamente 32 Kbytes de memória. O sistema OS/360, introduzido em 1964 pela IBM, incluía mais de 1 milhão de instruções de máquina. Ainda no início dos anos 70, o Multics, desenvolvido principalmente pelo MIT e pela Bell Labs, já possuía mais de 20 milhões de instruções de máquina. Atualmente mesmo

microcomputadores pessoais executam sistemas operacionais com elevada complexidade. O Windows NT 4.0 possui 16 milhões de linhas de código e o Windows 2000 mais de 30 milhões de linhas de código. Desde o início, sistemas operacionais foram considerados um dos tipos de software mais difíceis de serem construídos. Por exemplo, em [BOE 81] o autor classifica os sistemas em 3 tipos com o propósito de estimar seus custos, e os sistemas operacionais são incluídos na classe mais complexa.

Ao longo dos anos, muito esforço foi colocado no sentido de criar as estruturas de software mais apropriadas para organizar internamente os sistemas operacionais. Existem grandes diferenças entre os serviços oferecidos por um pequeno sistema operacional de propósito específico, embutido em um equipamento qualquer, e um sistema operacional de propósito geral como o Linux ou o Windows. Entretanto, as estruturas de software empregadas não variam tanto assim, mesmo entre sistemas cujo tamanho difere por 3 ordens de grandeza (milhares de linhas versus milhões de linhas). A taxonomia apresentada neste texto classifica a organização interna dos sistemas operacionais conforme a árvore mostrada na fig. 5.6. No restante desta seção cada uma das organizações que aparecem na árvore serão caracterizadas e discutidas.

As recomendações da engenharia de software valem também nesta área. O desenvolvimento de um sistema operacional não deixa de ser um projeto de software e sofre dos mesmos problemas que aplicações em geral. Por exemplo, a criação do OS/360 nos anos 60 envolveu cerca de 5000 programadores, o que obviamente não tolera práticas ad-hoc de desenvolvimento. Propriedades como modularidade, interfaces claras e a mais simples possível entre os módulos, alta coesão e baixo acoplamento são muito bem vindas.

4.3.1. *Kernel* monolítico

A forma mais simples de organizar um sistema operacional é colocar toda a sua funcionalidade dentro de um único programa chamado ***kernel***. O *kernel* inclui o código necessário para prover toda a funcionalidade do sistema operacional (escalonamento, gerência de memória, sistema de arquivos, protocolos de rede, *device-drivers*, etc). O *kernel* executa em modo supervisor e suporta o conjunto de chamadas de sistemas. Ele é carregado na inicialização do computador e permanece sempre na memória principal.

Internamente, o código do *kernel* é dividido em procedimentos os quais podem ser agrupados em módulos. De qualquer forma, tudo é ligado (*linked*) junto e qualquer rotina pode, a princípio, chamar qualquer outra rotina. Todas as rotinas e estruturas de dados fazem parte de um único espaço lógico de endereçamento. O conceito de processo existe fora do *kernel*, mas não dentro dele. Apenas um fluxo de execução existe dentro do *kernel*. Chama-se esse projeto de ***kernel* monolítico** (fig. 5.7). Uma interrupção de hardware ou de software gera um chaveamento de contexto do processo que estava executando para o fluxo interno do *kernel*, o qual possui seu próprio espaço de endereçamento e uma pilha para as chamadas de subrotinas internas.

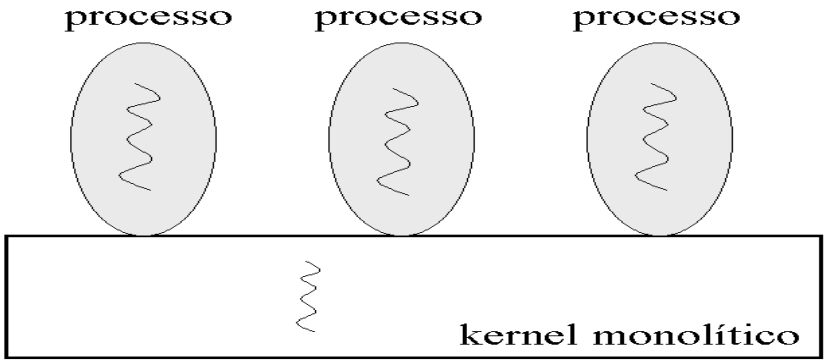

Figura 5.7: *Kernel* monolítico.

Duas melhorias podem ser feitas sem alterar a essência do *kernel* monolítico. Uma delas é a introdução de **programas de sistema**. Os programas de sistema são programas que executam fora do *kernel* (em modo usuário), fazem chamadas de sistema como programas normais, mas implementam funcionalidades típicas de sistemas operacionais, como listar os arquivos de um diretório ou gerenciar o compartilhamento de uma impressora (*spooler* de impressão). Uma vantagem de transferir parte da funcionalidade do *kernel* para os programas de sistema é a economia de memória. O *kernel* está sempre residente, enquanto programas de sistema são carregados para a memória somente quando necessário. Além disso, programas de sistema podem ser atualizados com facilidade, enquanto uma atualização do *kernel* exige uma reinstalação e reinicialização do sistema. Outra vantagem é com relação à questão da confiabilidade, pois uma falha em um programa de sistema dificilmente compromete os demais programas e o próprio *kernel*. Finalmente, programas de sistema geram uma modularização de qualidade que facilita o desenvolvimento e a manutenção do sistema.

Outra melhoria são os **módulos dinamicamente carregáveis**, isto é, permitir que algumas partes do *kernel* possam ser carregadas e removidas dinamicamente, sem interromper a execução do sistema. Tipicamente isto é feito para permitir que novos *device-drivers* sejam instalados sem a reinicialização do sistema. Memória é alocada para o *kernel*, o código do *device-driver* é carregado para esta memória, o endereço de suas rotinas é incluído em tabelas do *kernel*, e uma ligação dinâmica entre o módulo carregado e o resto do *kernel* é realizada. Mais recentemente este tipo de mecanismo foi ampliado para que outras funcionalidades possam também ser carregadas dinamicamente, como novos sistemas de arquivos. Entretanto, a maior parte do *kernel* e toda a sua funcionalidade básica possui carga estática.

Existem duas variações de *kernel* monolítico no que diz respeito às interrupções de hardware. Na forma mais simples, o *kernel* executa com interrupções desabilitadas. O ***kernel* monolítico não-interrompível** possui código mais simples, pois enquanto o *kernel* está executando nada mais acontece, nem mesmo interrupções de periféricos como temporizadores e controladores de disco. Programas de usuário executam com interrupções habilitadas todo o tempo. O preço a ser pago por esta simplificação é uma redução no desempenho. Por exemplo, se o controlador do disco termina o acesso em andamento e gera uma interrupção enquanto o *kernel* está executando, o disco ficará parado até que o *kernel* termine sua execução, retorne a executar código de usuário e

habilite as interrupções. Periféricos serão sub-utilizados, além de haver uma troca de contexto desnecessária, pois o processo de usuário ativado será interrompido.

Uma forma mais eficiente é executar o código do *kernel* com interrupções habilitadas. O ***kernel* monolítico interrompível** possui desempenho melhor pois os eventos associados com periféricos e temporizadores ganham imediata atenção, mesmo quando o código do *kernel* está executando. Entretanto, é preciso notar que tratadores de interrupções podem acessar estruturas de dados do *kernel*, as quais podem estar inconsistentes enquanto uma chamada de sistema é atendida. Nessa situação, a execução do tratador de interrupção poderia corromper todo o sistema. Na construção de um *kernel* monolítico interrompível é necessário identificar todas as estruturas de dados acessadas por tratadores de interrupção e, quando o código normal do *kernel* acessa essas estruturas de dados, interrupções devem ser desabilitadas (pelo menos aquelas cujos tratadores acessem a estrutura em questão). As estruturas de dados acessadas por tratadores de interrupção formam uma **seção crítica** que deve ser protegida, e o mecanismo usado para isto é simplesmente desabilitar as interrupções enquanto estas estruturas de dados estiverem sendo acessadas. É preciso cuidado quando as seções com interrupções desabilitadas aparecem aninhadas, pois neste caso somente ao sair da seção mais externa é que as interrupções podem ser habilitadas novamente.

Imagine agora a situação onde o código de um *kernel* monolítico interrompível está executando em função da chamada de sistema de um processo de baixa prioridade. Neste momento ocorre uma interrupção de tempo (*timer tick*) e um processo de alta prioridade é liberado. No *kernel* monolítico interrompível o tratador da interrupção executa, mas ao seu término o código do *kernel* volta a executar em nome de um processo de baixa prioridade, mesmo que um processo de alta prioridade aguarde na fila do processador. Tem-se então, que este tipo de *kernel* é intrinsecamente **não-premptável**, isto é, nenhum processo recebe o processador enquanto a linha de execução dentro do *kernel* não for concluída ou ficar bloqueada por alguma razão.

4.3.2. *Kernel* convencional

Chama-se de ***kernel* convencional** (fig. 5.8) aquele que, além de interrompível, permite uma troca de contexto mesmo quando código do *kernel* estiver executando. A passagem de um *kernel* monolítico interrompível para um *kernel* convencional possui várias implicações. Entre elas pode-se destacar que agora o conceito de processo existe também dentro do *kernel*, uma vez que o processo pode ser suspenso e liberado mais tarde enquanto executa código do *kernel*. No *kernel* monolítico apenas uma pilha basta, pois a cada momento apenas um fluxo de execução existe dentro dele. Interrupções podem acontecer, mas ainda assim uma única pilha é suficiente. No *kernel* convencional um processo pode passar o processador para outro processo que também vai executar código do *kernel*. Logo, é necessária uma pilha interna ao *kernel* para cada processo, além das pilhas em modo usuário.

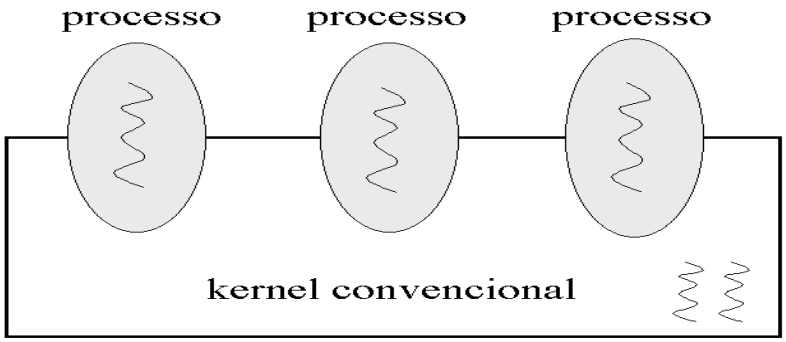

Figura 5.8: *Kernel* convencional.

Quando o processo executando código do usuário faz uma chamada de sistema, ocorre um chaveamento no modo de execução e no espaço de endereçamento, mas conceitualmente o mesmo processo continua executando. Apenas agora ele executa código do *kernel*. Este processo executa até que a chamada de sistema seja concluída. Nesse caso simplesmente é efetuado um retorno para o contexto do usuário. Também é possível que esse processo tenha que esperar por algum evento como, por exemplo, um acesso a disco. Nesse caso ele fica bloqueado e um outro processo passa a ser executado. Como vários processos podem estar bloqueados simultaneamente dentro do *kernel*, deve existir no espaço de endereçamento do *kernel* uma pilha para cada processo. Observe que, neste tipo de *kernel*, o processo executando código do *kernel* não perde o processador para outro processo, a não ser que fique bloqueado, caracterizando assim um ***kernel* convencional não-preemptável**.

No *kernel* convencional as interrupções de software e de exceção estão fortemente associadas com o processo em execução, representando uma chamada de sistema ou uma violação de algum tipo. Assim, as ativações dos tratadores desses dois tipos de interrupções são vistas apenas como uma nova fase na vida do processo. O mesmo não ocorre com as interrupções de hardware, as quais estão associadas, na maioria das vezes, com solicitações de entrada e saída feitas por outros processos.

No *kernel* monolítico, a chamada de sistema implica em chavear o modo de execução, o espaço de endereçamento e o próprio processo. No *kernel* convencional, a chamada de sistema implica tão somente no chaveamento do modo de execução e uma adaptação no espaço de endereçamento. Se acontecer um retorno imediato da chamada de sistema (por exemplo com `gettime()` ou `read()` de dados bufferizados), os outros dois tipos de chaveamento sequer acontecerão. O resultado final é a melhoria no desempenho do sistema, embora o código do *kernel* fique um pouco mais complexo. Em alguns sistemas esta solução é dita empregar um processo envelope, pois os processos "envelopam" (executam) tanto o programa usuário como o código do sistema operacional [HOL 83]. É possível a existência de processos que jamais executam código fora do *kernel*, isto é, processos que executam tarefas auxiliares dentro do *kernel* e não estão associados com nenhum programa de usuário em particular. Este tipo de processo pode ser usado para, por exemplo, escrever blocos de arquivos alterados para o disco.

O código do *kernel* convencional pode ser visto como um conjunto de rotinas, possivelmente agrupadas em módulos, que estão à disposição dos processos. Obviamente os usuários estão sujeitos ao controle de acesso embutido no *kernel*, o qual é preservado pela MMU, que impede o acesso direto à memória do *kernel* por parte de

programas de usuário. Muitas vezes o espaço de endereçamento de um processo inclui tanto os segmentos com código e dados do programa de usuário quanto o código e dados do *kernel*. Mecanismos de proteção associados com o modo de execução impedem que o código executado em modo usuário acesse diretamente os *bytes* pertencentes aos segmentos do sistema.

Quando o processo dentro do *kernel* deve ser bloqueado, ocorre um chaveamento entre processos. Algumas rotinas do *kernel* são responsáveis por este efeito e, quando elas executam, conceitualmente nenhum processo está executando. Tem-se a transição:

- Processo P1 executando código do *kernel*.
- Processo P1 chama rotina que executa chaveamento entre processos.
- Rotina de chaveamento salva todo o contexto do processo P1.
- Processo P1 transformou-se em um fluxo de execução interno ao *kernel*, não vinculado a nenhum processo, cujo único propósito é carregar o próximo processo a executar.
- Rotina de chaveamento de processos carrega todo o contexto do processo P2, o fluxo de execução transforma-se no processo P2.
- Processo P2 retoma a execução do ponto onde havia sido bloqueado.

A distinção entre *kernel* monolítico interrompível e *kernel* convencional não-preemptável pode ser resumida pela execução do código do *kernel* estar associada com processos ou com um fluxo desvinculado do conceito de processo. Na prática isto pode ser determinado pelo número de pilhas internas ao *kernel*. Se existe somente uma pilha dentro do *kernel*, tem-se um *kernel* monolítico interrompível. Já o *kernel* convencional, mesmo não-preemptável, exige uma pilha dentro do *kernel* para cada processo. Ao mesmo tempo, as rotinas do *kernel* convencional devem ser **re-entrantes**, isto é, devem ser programadas de tal forma que um processo possa iniciar a execução, ser suspenso por algum motivo, e a mesma rotina ser então executada por outro processo. O *kernel* agora corresponde realmente a um programa concorrente, algo que não acontecia com o *kernel* monolítico.

Como o *kernel* convencional é um programa concorrente, as suas estruturas de dados devem ser protegidas, pois formam seções críticas dentro do programa. A sincronização entre processos dentro do *kernel* é feita de várias formas. Primeiramente, no *kernel* convencional não-preemptável o processo só libera o processador voluntariamente. Desta forma, a maioria das estruturas de dados não precisa ser protegida pois é garantido, pela disciplina de programação, que o processo fazendo o acesso deixará a estrutura em um estado consistente antes de liberar o processador. Quando, em função dos algoritmos usados, o processo necessita bloquear-se ainda com uma estrutura de dados em estado inconsistente, alguma primitiva de sincronização deve ser usada, como semáforos ou algo semelhante. Se outro processo chamar o *kernel* e tentar acessar esta estrutura de dados que está inconsistente, então ele ficará bloqueado no semáforo. Finalmente, estruturas de dados também acessadas pelos tratadores de interrupção de hardware somente podem ser acessadas por um processo com as interrupções desabilitadas, como já acontecia antes no *kernel* monolítico.

Outra questão relevante, relacionada com as estruturas de dados internas ao *kernel*, leva ao surgimento de mais dois tipos de *kernel* convencionais. O ***kernel* convencional com pontos de preempção** somente suspende um processo que executa código do *kernel* em pontos previamente definidos do código, nos quais é sabido que nenhuma estrutura de dados está inconsistente. O desempenho deste tipo de *kernel* é superior ao *kernel* monolítico, mas o processo de mais alta prioridade ainda é obrigado a

esperar até que a execução do processo de mais baixa prioridade atinja um ponto de preempção. O sistema SVR4.2/MP funciona assim.

Por outro lado, o ***kernel* convencional preemptável** realiza o chaveamento de contexto tão logo o processo de mais alta prioridade seja liberado. Para isto, todas as estruturas de dados do *kernel* que são compartilhadas entre processos devem ser protegidas por algum mecanismo de sincronização, como semáforos, mutexes, etc. Esta solução, usada no Solaris 2.x, resulta em melhor resposta do sistema aos eventos externos, e um comportamento mais coerente com as prioridades dos processos.

Uma vez que existe no *kernel* convencional uma estrutura disponível para a convivência de vários fluxos de execução simultâneos internos ao *kernel*, pode-se utilizar construções denominadas de *threads* do *kernel*. As ***threads* do *kernel*** são fluxos de execução que executam o tempo todo código do *kernel* e não possuem nenhum código de usuário associado. Elas executam tarefas auxiliares, como a manutenção dos níveis de memória disponível através da escrita de *buffers* para o disco ou implementam protocolos de comunicação em rede.

Alguns sistemas como o Solaris [VAH 96] vão além, e transformam os tratadores de interrupção em *threads* de *kernel*. Dessa forma, a única coisa que o tratador de interrupção propriamente dito faz é liberar a *thread* de *kernel* correspondente, inserindo-a na fila do processador. O código executado pela *thread* é que realiza efetivamente o tratamento da interrupção. A sincronização com respeito às estruturas de dados do *kernel* é feita normalmente, pois tanto processos normais como *threads* de *kernel* podem ser bloqueados se isto for necessário. O código fica mais simples e robusto, pois agora não é necessário determinar exatamente qual estrutura de dados é acessada por qual tratador de interrupção, para desabilitar as interrupções corretamente durante o acesso. Simplesmente cada estrutura de dados compartilhada é protegida por um semáforo ou semelhante e o problema está resolvido. A experiência com o Solaris também mostra que não existe redução de desempenho quando este esquema é utilizado, pois um conjunto de threads pode ser previamente criado, para evitar o custo da criação a cada interrupção.

4.3.3. Microkernel

Recentemente uma organização tem sido proposta na qual a funcionalidade típica de *kernel* é dividida em duas grandes camadas. Esta solução baseia-se na existência de um ***microkernel***, o qual suporta os serviços mais elementares de um sistema operacional: gerência de processador e uma gerência de memória simples. A idéia de *microkernel* foi popularizada pelo sistema operacional Mach [TAN 95].

Na sua forma mais simples, o *microkernel* corresponde à funcionalidade de mais baixo nível do *kernel* convencional preemptável, aquela responsável por realizar o chaveamento dos processos envelopes. O *microkernel* suporta um *kernel* semelhante ao convencional preemptável, separado da funcionalidade inserida no *microkernel*. Esta solução é ilustrada pela fig. 5.9 e será chamada de ***microkernel* com *kernel***.

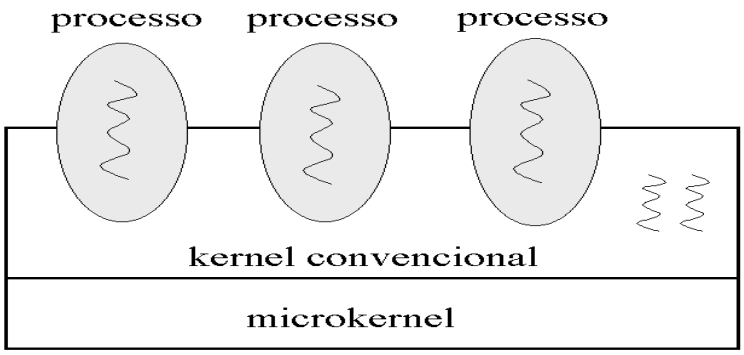

Figura 5.9: *Microkernel* com *kernel*.

Um passo adiante na idéia de *microkernel* é criar um conjunto de processos servidores autônomos, tal que as chamadas de serviço dos processos de usuário são feitas diretamente, sem passar por uma camada que uniformize a interface do *kernel*. Nesta solução tem-se um *microkernel* que suporta um conjunto de processos com espaços de endereçamento independentes. Alguns processos executam programas de usuário enquanto outros implementam serviços do sistema operacional, tais como o sistema de arquivos e o gerenciador de memória virtual. Quando o processo do usuário necessita ler um arquivo ele envia uma mensagem para o processo "sistema de arquivos", o qual executa o pedido e retorna outra mensagem com o resultado. A participação do *microkernel* resume-se a dividir o tempo do processador entre os diversos processos e prover o mecanismo de comunicação entre eles. Internamente, o *microkernel* tem a forma de um pequeno *kernel* monolítico. As desvantagens do *kernel* monolítico não são tão sérias neste caso pois o código do *microkernel* é pequeno. Por sua vez, cada processo servidor constitui um espaço de endereçamento independente e pode ser composto por várias *threads* para melhorar o seu desempenho. Chama-se essa organização de ***microkernel* com processos servidores** (fig. 5.10).

Os processos servidores que necessitam acesso especial podem executar em modo supervisor. O modo de execução passa a ser uma característica do processo, definida pelo administrador do sistema. Modos de execução intermediários podem ser utilizados se estiverem disponíveis. O *microkernel* sempre executa em modo supervisor e pode ser visto como um componente monolítico. Um sistema operacional bastante conhecido que trabalha dessa forma é o Minix [TAN 00].

A existência de processos servidores representa uma decomposição funcional que facilita o desenvolvimento do sistema operacional nos aspectos de codificação, teste e manutenção. As vantagens da organização baseada em *microkernel* são semelhantes àquelas dos programas de sistema: facilidade para atualizações, isolamento entre os diferentes componentes, depuração e manutenção mais fácil, modularização superior, facilidade para distribuição e tolerância a falhas. Também, agora, tem-se o *microkernel* e os processos servidores executando em espaço de endereçamento diferentes, o que é ótimo do ponto de vista da engenharia de software, embora represente uma penalidade no tempo de execução. Isto resulta em facilidade para desenvolver e testar novos processos servidores. Também o sistema como um todo responde melhor às definições de prioridades feitas pela administração, pois a preempção ocorre mais facilmente.

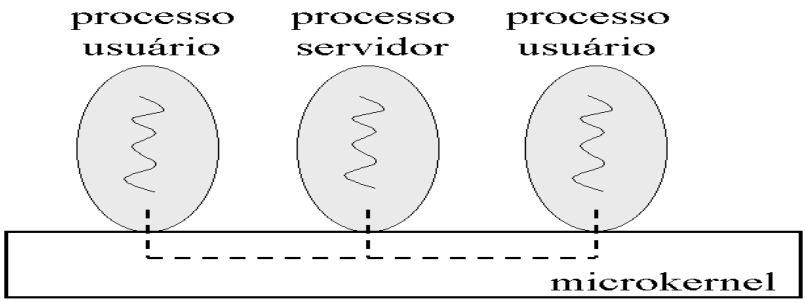

Figura 5.10: *Microkernel* com processos servidores.

A grande motivação para empregar uma organização como esta é a modularidade e flexibilidade resultantes. É possível configurar cada instalação com apenas os processos servidores desejados, depois atualizá-los sem a necessidade de reinicializar o sistema. É especialmente atraente em sistemas distribuídos, pois os mecanismos utilizados para implementar serviços distribuídos são complexos e, se colocados todos em um *kernel* convencional, o tornariam intratável. Nesta organização pode-se com mais facilidade distribuir os processos por diferentes computadores e ainda assim obter elevado nível de transparência. Apenas o *microkernel* deve executar em todos os computadores, já que não existe memória compartilhada entre os processos servidores e os processos usuários.

Uma das mais notáveis capacidades dos sistemas baseados em *microkernel* é suportar simultaneamente várias API de diferentes sistemas operacionais. Por exemplo, é possível colocar sobre um mesmo *microkernel* processos servidores que suportem as chamadas da API Win32 e outros processos servidores que suportem a API Posix. Embora todo este código de sistema operacional represente uma carga considerável para o computador, em determinados ambientes esta possibilidade é atrativa. Esta capacidade pode também ser implementada sobre outros tipos de organização, mas não com a mesma facilidade.

A maior desvantagem dessa solução é o custo muito mais alto associado com uma chamada de sistema, que agora implica em vários chaveamentos sucessivos. Além disso, o programa de usuário deve montar e desmontar mensagens ao solicitar serviços para outros processos, pois mensagens formam a base da comunicação neste tipo de organização. Cada passagem de nível (processo de usuário para *microkernel* para processo de sistema e tudo de volta) implica no gasto de tempo do processador. Por exemplo, a comunicação entre sistema de arquivos e gerência de memória era feita antes com uma chamada de subrotina, mas agora exige uma chamada ao *microkernel*.

Quanto menor a funcionalidade do *microkernel*, maior a penalidade no desempenho. Assim, uma tendência atual tem sido a re-incorporação de servidores importantes ao *microkernel*. Ele passa a ser um pouco "menos micro", perdendo em parte a flexibilidade, mas o desempenho melhora. Isto é conseguido através da redução do número de chaveamentos de modo de execução e de espaços de endereçamento no atendimento das chamadas de sistema. Os serviços que são os maiores candidatos a serem incluídos no *microkernel* são a gerência de memória e os *device-drivers*, pelo menos aqueles associados com os principais periféricos do sistema.

Esta questão do desempenho resultou em variações na literatura a respeito de onde está exatamente a fronteira entre um *kernel* e um *microkernel*. Por exemplo, quais

os serviços que devem ser implementados ou qual o tamanho máximo que um *microkernel* pode ter. Entretanto, é mais ou menos consenso que, minimamente, o *microkernel* deve oferecer processos (e *threads*), comunicação entre processos, facilidades para a gerência de entrada/saída e interrupções e gerência de memória física (de baixo nível).

O mecanismo tipicamente usado entre processos acima do *microkernel* é a troca de mensagens. Uma das motivações básicas dessa organização é a independência entre os processos servidores, e a definição de variáveis compartilhadas prejudicaria esse propósito. O *microkernel* suporta primitivas elementares de envio e recepção de mensagens, as quais são usadas pelos processos de usuário para solicitar serviços aos processos servidores e por estes para enviar as respectivas respostas. As variações típicas dos mecanismos de mensagens, como endereçamento direto ou indireto, mensagens tipadas ou não, grau de bufferização, semântica das primitivas, também são encontradas aqui. Para que o *microkernel* cumpra seu papel de isolar os diferentes processos, ele deve ser capaz de controlar os direitos de acesso e a integridade das mensagens. Para evitar que o envio de uma mensagem implique na cópia memória-memória, operação tipicamente lenta, mecanismos como **remapeamento de páginas** podem ser usados. Neste caso, a mensagem não é realmente copiada, mas a página física que ela ocupa é mapeada na tabela de páginas do processo destinatário.

Uma questão importante é como o *microkernel* deve tratar a ocorrência de uma interrupção. Algumas interrupções, como as dos temporizadores do hardware (*timers*) e aquelas associadas com erros de execução (acesso ilegal a memória, instrução ilegal, etc) possuem um tratamento óbvio. A questão refere-se às interrupções geradas por periféricos cuja gerência é realizada por processos servidores que executam fora do *microkernel*. Uma solução simples é permitir que processos servidores instalem suas rotinas de tratamento de interrupções. Uma chamada de *microkernel* permite ao processo servidor informar que, na ocorrência de interrupção do tipo "x", a rotina "y" do processo servidor deve ser chamada. Este esquema pode ser usado em sistemas menores, mas possui várias desvantagens. Por exemplo, se memória virtual for empregada é possível que a rotina em questão sequer esteja residente na memória principal. Além disso, a programação do processo servidor fica mais complexa, pois além das mensagens dos processos clientes ele deve se preocupar agora com rotinas executadas automaticamente. Entretanto, este tipo de programação é comum no mundo Unix, quando o mecanismo de sinais é utilizado.

Uma solução mais harmoniosa com a organização geral do sistema é fazer o *microkernel* transformar as interrupções em mensagens. Inicialmente, o processo servidor interessado nas interrupções do tipo "x" informa isto ao *microkernel*. Sempre que ocorrer uma interrupção deste tipo, o *microkernel* gera automaticamente uma mensagem para o processo servidor em questão, informando que uma interrupção daquele tipo aconteceu. A programação do processo servidor é simplificada, pois ele agora sempre recebe mensagens, seja dos demais processos, seja do dispositivo de hardware que ele controla. Obviamente, existe uma penalidade em termos de desempenho na montagem e envio dessa mensagem, se comparada com a solução onde uma rotina do processo servidor é chamada diretamente.

O aspecto mais complexo no projeto de um *microkernel* está exatamente na gerência da memória. Para que os processos executando acima do *microkernel* possuam espaços de endereçamento independentes, o *microkernel* deve criar e manter as respectivas tabelas de páginas e de segmentos, conforme a arquitetura do processador. Ele também deve gerenciar quais páginas físicas estão livres na memória. Entretanto, o mecanismo de memória virtual envolve interações com o disco e deveria, a princípio,

ficar fora do *microkernel*. Quando um processo executa um acesso ilegal a memória o *microkernel* pode recorrer a um processo servidor para realizar o tratamento da falta de página, com possível acesso a disco e escolha de página vítima. Entretanto, a API do *microkernel* para este processo servidor exige um projeto cuidadoso, ou sérias penalidades no desempenho acontecerão.

Sistemas baseados em *microkernel* podem variar com respeito aos modos de execução. A maioria dos processadores oferecem pelo menos dois modos de execução: supervisor e usuário. No caso de mais de dois denomina-se os demais como modos de execução intermediários, com mais direitos que o modo usuário mas nem todos os direitos do modo supervisor. Por exemplo, o Intel Pentium oferece 4 níveis de proteção: o nível 0 corresponde ao modo supervisor, nível 3 é o modo usuário típico e os níveis 1 e 2 oferecem direitos intermediários.

Se o sistema está baseado em *microkernel*, naturalmente o código de usuário executa em modo usuário e o código do *microkernel* executa em modo supervisor. Entretanto, o código dos servidores permite várias possibilidades. Buscando segurança é possível executar o código dos servidores em modo usuário, obrigando a realização de uma chamada ao *microkernel* toda vez que uma operação restrita for necessária. Esta solução gera vários chaveamentos de modo de operação e a execução de código extra pelo *microkernel* para testar a validade das operações solicitadas. Com o propósito de tornar o sistema mais rápido, é possível executar o código dos servidores em modo supervisor. O sistema fica mais simples e rápido, porém agora qualquer servidor pode corromper todo o sistema e a proteção que havia antes, mesmo entre servidores, é perdida. Idealmente o código do sistema operacional não deveria conter erros, mas a nossa experiência diária mostra que isto não é sempre verdade. Por fim, modos de execução intermediários podem ser usados para os servidores, buscando uma solução de compromisso. Por exemplo, permitir que processos servidores acessem alguns endereços de entrada e saída previamente estabelecidos, o que não é possível para processos que executam em modo usuário.

Sistemas baseados em *microkernel* também podem variar com respeito aos espaços de endereçamento. Logicamente, o código de usuário executa em espaço de endereçamento próprio, o mesmo acontecendo com o código do *microkernel*. Entretanto, o código de cada servidor pode executar em espaço de endereçamento independente ou ser ligado ao código do *microkernel*, ocupando portanto o mesmo espaço de endereçamento. Novamente, tem-se uma questão de isolamento (proteção) versus desempenho. O chaveamento entre espaços de endereçamento exige comandos para a MMU (memory management unit) e também esvaziamento de sua *cache* interna, a TLB (*translation lookaside buffer*). Ao juntar *microkernel* e servidores no mesmo espaço de endereçamento, não é necessário chavear o contexto da MMU quando o servidor faz uma solicitação de serviço ao *microkernel*. Entretanto, o isolamento já não é imposto pela MMU, o que diminui o isolamento entre os diferentes componentes do sistema operacional. Também fica mais complexa a carga dinâmica de servidores.

Concluindo, sistemas do tipo *microkernel* com *kernel* podem ser considerados uma simplificação da idéia geral de *microkernel*. Eles têm uma organização modular, com interfaces claras entre os seus componentes. Dados essenciais são encapsulados e somente podem ser acessados através das rotinas apropriadas. Entretando, vários servidores importantes executam em modo supervisor no mesmo espaço de endereçamento, formando o *kernel* sobre o *microkernel*. Esta organização busca melhorar o desempenho do sistema ao reduzir a necessidade de chaveamentos de contextos e modos de execução. Existe aqui portanto uma variação da idéia de *microkernel*, onde todos os processos servidores são agrupados em um componente de

software, o qual adiciona uma interface para as chamadas de sistema. Infelizmente, a terminologia, que já varia no mundo acadêmico, varia ainda mais no mundo comercial. Como os departamentos de marketing das empresas determinaram que "*microkernel* é bom", agora todos os desenvolvedores de sistemas operacionais tentam mostrar que, de alguma forma, o seu sistema também é baseado em *microkernel*.

4.3.4. Comparação entre as organizações

As técnicas apresentadas nas seções anteriores são as fundamentais, mas nada impede que um sistema operacional seja construído misturando vários aspectos aqui descritos. Por exemplo, o conceito de processo servidor pode ser usado sobre um *kernel* monolítico para suportar determinados serviços com tempo de resposta mais dilatado. O *microkernel* em si pode ser visto como um *kernel* monolítico com pouquíssimos serviços. Cada sistema operacional existente utiliza uma mistura dessas técnicas, em geral aproximando-se mais de uma delas. As estruturas de software apresentadas neste texto devem ser entendidas como os modelos abstratos (as receitas de bolo) utilizados na construção dos sistemas operacionais, embora cada sistema (cada bolo) tenha sempre as suas peculiaridades e características próprias. Como regra geral, sistemas acadêmicos buscam a elegância e a qualidade no projeto, e portanto tendem a usar *microkernel*. Por outro lado, sistemas comerciais buscam principalmente desempenho, usando por isto soluções baseadas em *kernel* convencional. De qualquer forma, todos os sistemas apresentam uma mistura das formas básicas, repletas de detalhes específicos, que vão além dos objetivos deste texto.

Como em qualquer tipo de software, a chave para entender a organização interna de um sistema operacional está na compreensão de como ele foi decomposto em diversas partes. A decomposição pode ser analisada segundo diferentes perspectivas, mas quatro delas destacam-se, para os propósitos deste texto.

Primeiramente, tem-se a decomposição do código em **rotinas e módulos**. Isto é feito para encapsular variáveis locais, tanto a nível de rotina como de módulo, e também rotinas internas ao seu módulo, as quais não podem ser chamadas de outros módulos. Orientação a objetos vai nessa mesma direção, embora ainda não seja empregada completamente na construção de *kernel*. Este particionamento é efetuado pelo compilador e pelo ligador, durante a tradução do código fonte do sistema operacional. A divisão rígida do sistema em camadas não é normalmente suportada por compiladores e ligadores, dependendo da sua adoção como disciplina de projeto e de programação.

Uma decomposição óbvia está relacionada com os direitos de acesso de cada parte, através dos **modos de execução do processador**. Considerar-se-á, aqui, a existência de apenas dois modos de execução (supervisor e usuário), embora outros intermediários também possam existir. Este particionamento é decidido em projeto e implementado pela unidade de controle do processador.

O sistema operacional também pode ser decomposto em vários **espaços de endereçamento**. Neste caso, a visibilidade das rotinas e estruturas de dados do sistema operacional pelas outras partes do mesmo pode ser limitada através da criação de vários espaços de endereçamento independentes. Este particionamento é efetuado pela MMU.

Finalmente, tem-se a decomposição da execução do sistema operacional em vários **fluxos de execução** separados, os quais provavelmente colaboram entre si, tornando o sistema operacional um programa concorrente. O termo fluxo de execução é usado para designar *threads*, processos, ou quaisquer construções semelhantes. Este particionamento é efetuado por uma decisão de projeto e disciplina de programação.

Uma vez definidas as quatro perspectivas da decomposição de um sistema operacional que serão analisadas aqui, pode-se investigar como cada tipo de organização decompõe o sistema operacional nessas perspectivas. Embora o texto tenha indicado a existência de variações a partir das organizações básicas, a análise considera apenas as 3 principais formas de organização: *kernel* monolítico interrompível, *kernel* convencional preemptável e *microkernel* com processos servidores.

A **decomposição em rotinas e módulos** pode ser aplicada a qualquer forma de organização. A modularização é um dos pilares da computação e pode ser aplicada a qualquer tipo de software. Até mesmo no *microkernel*, embora pequeno, a modularização vai melhorar a legibilidade e robustez, facilitando testes e manutenção.

A **decomposição em modos de execução** é bem característica em cada forma de organização:

- No *kernel* monolítico interrompível os processos de usuário executam em modo usuário, enquanto o *kernel* executa em modo supervisor.

- No *kernel* convencional preemptável o processo executa código de usuário em modo usuário e, após uma chamada de sistema, executa código do *kernel* em modo supervisor. Os tratadores de interrupção fazem parte do *kernel* e executam em modo supervisor. Durante o chaveamento entre processos são executadas instruções fora do contexto de qualquer processo. Este pequeno fluxo de execução interno ao *kernel* também executa em modo supervisor.

- No *microkernel* com processos servidores os processos de usuário executam código de usuário em modo usuário e o *microkernel* executa em modo supervisor. Os processos servidores podem executar em modo usuário, modo supervisor ou algum modo intermediário, dependendo do tipo de serviço prestado e do projeto do sistema.

A **decomposição em espaços de endereçamento** também caracteriza com força as diferentes organizações:

- No *kernel* monolítico interrompível cada processo de usuário possui um espaço de endereçamento próprio, enquanto o *kernel* executa em seu próprio espaço de endereçamento, o qual permite acesso a toda a memória.

- No *kernel* convencional preemptável existem algumas possibilidades. Em essência cada processo possui um espaço de endereçamento quando executa código de usuário e passa para outro espaço de endereçamento quando executa código do *kernel*. Entretanto, uma única tabela de páginas e/ou segmentos pode ser usada para os dois espaços de endereçamento, sendo a distinção obtida através do modo de execução, ou seja, as partes da memória associadas com o *kernel* somente podem ser acessadas quando o processo executa em modo supervisor, isto é, executa código do *kernel*. Neste caso, uma chamada de sistema tem o efeito de mudar o processador para modo supervisor, liberar as partes do *kernel* nas tabelas de memória e desviar a execução para o código do *kernel*. Tratadores de interrupção também executam neste espaço de endereçamento ampliado.

- No *microkernel* com processos servidores cada processo, seja de usuário seja servidor, executa em um espaço de endereçamento próprio. O *microkernel* também possui seu próprio espaço de endereçamento, permitindo acesso a toda a memória.

A **decomposição em fluxos de execução** também caracteriza muito bem as diferentes soluções de organização usadas em sistemas operacionais:

- No *kernel* monolítico interrompível cada processo de usuário corresponde a um fluxo de execução. Se *threads* são suportadas, cada processo de usuário pode conter vários fluxos de execução. O *kernel*, por sua vez, é composto por um único fluxo de execução principal, associado com uma chamada de sistema.

- No *kernel* convencional preemptável cada processo está associado com um fluxo de execução. Se *threads* são suportadas então cada processo estará associado com vários fluxos de execução. Estes fluxos executam tanto código de usuário quanto o código do *kernel*. Durante o chaveamento de contexto entre processos existe um momento no qual a execução está desvinculada de qualquer processo e pode ser pensada como um fluxo do *kernel* que aparece somente neste momento. Finalmente, tratadores de interrupção representam fluxos disparados por interrupções do hardware e encerrados ao término da rotina de tratamento.

- O *microkernel* com processos servidores tipicamente suporta *threads*. Logo, cada processo, seja de usuário ou servidor, pode estar associado com vários fluxos de execução. O *microkernel* pode ser implementado de várias formas, mas na mais simples ele seria apenas um núcleo monolítico não interrompível, caracterizado portanto por apenas um fluxo de execução. Na verdade esta descrição pode tornar-se recursiva, na medida que aplicarmos também ao *microkernel* as várias soluções possíveis descritas para o *kernel* principal.

Para completar a comparação entre as 3 soluções básicas, considerar-se-á, agora, o que acontece quando um programa de usuário faz uma chamada de sistema `read()` para ler dados de um arquivo previamente aberto. Na maioria das linguagens de programação o programador chama uma rotina da biblioteca da linguagem que, por sua vez, chama o sistema operacional. Em função do propósito deste texto, ignora-se a biblioteca da linguagem e iniciar a análise a partir da chamada ao sistema operacional propriamente dita. Também será suposto que este `read()` não pode ser satisfeito com dados na memória principal e um acesso ao disco será necessário.

No *kernel* monolítico interrompível o processo deve colocar os parâmetros nos locais apropriados e executar a interrupção de software associada com chamadas de sistema. Em função da interrupção de software a execução desvia para o código do *kernel*, executando em modo supervisor e com um novo espaço de endereçamento. Inicialmente o contexto de execução do processo chamador é salvo. É feito o processamento da chamada até o ponto no qual o pedido de acesso a disco foi enfileirado e nada mais resta a fazer. Neste momento um outro processo é selecionado para execução. Mais tarde uma interrupção do controlador do disco vai sinalizar a conclusão da leitura. O tratador desta interrupção vai liberar o processo chamador, que volta a disputar o processador. É sem dúvida a organização mais simples. Sua limitação está em permitir que um processo de baixa prioridade monopolize o processador enquanto executa código do *kernel*.

No *kernel* convencional preemptável o processo também deve colocar os parâmetros nos locais apropriados e executar a interrupção de software associada com chamadas de sistema. Em função da interrupção de software a execução desvia para o código do *kernel*, executando em modo supervisor e com um espaço de endereçamento ampliado. Entretanto, conceitualmente, o mesmo processo está executando. É feito o processamento da chamada até o ponto no qual o pedido de acesso a disco foi enfileirado e nada mais resta a fazer. Neste momento o processo declara-se bloqueado e solicita um chaveamento de processo. Seu contexto é salvo e o contexto de outro processo da fila de aptos é carregado. Observe que durante a transição a execução prossegue sem estar conceitualmente associada com nenhum processo em particular. Esta transição é crítica e acontece com interrupções desabilitadas. Mais tarde, a interrupção do controlador do disco sinaliza a conclusão da operação e o seu tratador vai liberar o processo chamador, colocando-o novamente na fila de aptos. Depois de algum tempo ele é eleito para execução, a qual é retomada dentro do *kernel*, no ponto imediatamente posterior àquele onde ele solicitou o seu bloqueio. O processamento da

chamada de sistema é concluído e este processo finalmente retorna para o código de usuário, restaurando modo de execução e espaço de endereçamento.

Nesta organização, caso uma interrupção de hardware libere um processo de mais alta prioridade, o mesmo retoma sua execução, mesmo que outro processo de mais baixa prioridade esteja executando código do *kernel*. Esta solução melhora o comportamento do sistema, se comparada com o *kernel* monolítico, pois ela respeita mais as diferentes prioridades dos processos. Entretanto, sua programação é mais complexa, pois agora todo o *kernel* tornou-se um enorme programa concorrente.

No *microkernel* com processos servidores o processo de usuário monta uma mensagem com a requisição de um `read` e seus parâmetros. Ele então solicita ao *microkernel* que envie esta mensagem para o processo servidor "sistema de arquivos" através do serviço `send()`. Isto implica em uma interrupção de software e a correspondente execução do código do *microkernel*. O *microkernel* verifica a integridade e validade da mensagem e a insere na fila de mensagens endereçadas para o processo servidor. Provavelmente o processo servidor "sistema de arquivos" possui uma prioridade maior que processos de usuário e, caso ele esteja bloqueado por uma chamada `receive()` anterior, ele será liberado. Neste caso o `send()` do processo usuário acaba resultando em sua preempção por processo de mais alta prioridade. Seja como for a mensagem será transferida para o processo servidor onde provavelmente uma de suas *threads* atenderá a requisição. Possivelmente o *device-driver* do disco em questão é implementado por outro processo servidor, e a mesma seqüência de eventos acontecerá novamente, agora com o servidor "sistema de arquivos" no papel de cliente do servidor "*device-driver* do disco". As respostas para as requisições implicam novamente em mensagens, dessa vez fazendo o caminho inverso. No final, tem-se 3 processos envolvidos, 2 envios de mensagem com requisição e 2 envios de mensagem com resposta, o que implica em 8 chamadas ao *microkernel* (4 *send* e 4 *receive*). Este tipo de organização exige cuidado na implementação para que o desempenho não fique muito abaixo daquele obtido com um *kernel* convencional.

4.4. Outras formas de organização

De maneira ortogonal às organizações fundamentais apresentadas na seção anterior, existem técnicas que podem ser aplicadas no sentido de minimizar a complexidade do projeto ou permitir a adaptação das organizações básicas a contextos diferentes. Abaixo são analisadas 3 dessas técnicas: a organização em camadas, o emprego de máquinas virtuais e a adaptação do *kernel* para máquinas multiprocessadoras.

4.4.1. Organização em camadas

Uma das primeiras técnicas, além da modularidade, usadas para lidar com a complexidade dos sistemas operacionais foi a organização hierárquica. Nesta estrutura o sistema é dividido em camadas. Cada camada utiliza os serviços da camada inferior e cria novos serviços para a camada superior. Idealmente, a implementação de uma camada pode ser livremente substituída por outra, desde que a nova implementação suporte a mesma interface e ofereça os mesmos serviços. Esta idéia apareceu pela primeira vez no sistema operacional THE (Technische Hogeschool Eindhoven, [DIJ 68]), criado por Dijkstra no final dos anos 60. Mais tarde esta mesma solução foi usada pela ISO (International Standards Organization) na famosa organização em 7 camadas

do seu modelo de referência para redes de computadores, o OSI - Open Systems Interconnection.

O sistema operacional THE foi organizado em 5 camadas. Na camada 0 ficava o hardware. Na camada 1 ficava o escalonamento do processador e a criação do conceito de processo. A camada 2 correspondia à gerência de memória e era responsável pela implementação de memória virtual. A camada 3 continha o *device-driver* para o console do operador. A camada 4 implementava o sistema de *buffers* para os demais dispositivos de entrada e saída e, por estar na camada 4, podia usar a memória virtual implementada pela camada 3 e enviar mensagens para o operador pela camada 2. Finalmente, programas de usuário executavam na camada 5.

Em geral não é fácil compor uma pilha de camadas de tal sorte que cada camada utilize somente as camadas inferiores. Veja, por exemplo, o relacionamento entre sistema de arquivos e gerência de memória. O sistema de arquivos faz chamadas para a gerência de memória solicitando páginas físicas para implementar um esquema de *cache* para os arquivos. Por outro lado, a gerência de memória pode solicitar uma leitura de arquivo quando ocorrer uma falta de página. A estrutura em camadas que o sistema deve ter para satisfazer a todas as necessidades não é óbvia.

4.4.2. Máquinas virtuais

É interessante notar que idéias que parecem ser novas nem sempre são tão novas assim. No início da década de 70 a IBM oferecia um produto chamado CP (Control Program, [SIL 99]) que criava, sobre o hardware do computador IBM 370, a ilusão de várias máquinas virtuais. Sobre cada máquina virtual era possível executar um sistema operacional completo.

Outro sistema que segue a filosofia das máquinas virtuais, pelo menos parcialmente, é Real-Time Linux. O RT-Linux (http://www.rtlinux.org) é uma extensão do Linux que se propõe a suportar tarefas com restrições temporais críticas. O seu desenvolvimento iniciou no "Department of Computer Science" do "New Mexico Institute of Technology". Atualmente o sistema é mantido pela empresa FSMLabs e já está em sua terceira versão.

O RT-Linux é um sistema operacional no qual um *microkernel* de tempo real coexiste com o *kernel* do Linux. O objetivo deste arranjo é permitir que aplicações utilizem os serviços sofisticados e o bom comportamento no caso médio do Linux tradicional, ao mesmo tempo em que permite que tarefas de tempo real operem sobre um ambiente mais previsível e com baixa latência. O *microkernel* de tempo real executa o *kernel* convencional como sua tarefa de mais baixa prioridade (Tarefa Linux), usando o conceito de máquina virtual para tornar o *kernel* convencional e todas as suas aplicações completamente interrompíveis.

Todas as interrupções são inicialmente tratadas pelo *microkernel* de tempo real, e são passadas para a tarefa Linux somente quando não existem tarefas de tempo real para executar. Para minimizar mudanças no *kernel* convencional, o hardware que controla interrupções é emulado. Assim, quando o *kernel* convencional "desabilita interrupções", o software que emula o controlador de interrupções passa a enfileirar as interrupções que acontecerem e não forem completamente tratadas pelo *microkernel* de tempo real. Pode-se dizer que o *microkernel* controla o mecanismo real de interrupções do processador, enquando o *kernel* Linux manipula um sistema virtual de interrupções, suportado pelo *microkernel* através da manipulação de interrupções reais.

4.4.3. Multiprocessamento simétrico

As colocações feitas nas seções anteriores consideraram um sistema operacional para máquinas com um único processador. No caso de uma **máquina SMP** (*symmetric multiprocessing*) outros mecanismos devem ser acrescentados. Na arquitetura SMP todos os processadores são tratados igualmente, não existe a figura de um processador principal ou mestre (por isto o termo "simétrico"). Todos os processadores compartilham a mesma memória e um dispositivo no hardware chamado de árbitro de barramento evita os conflitos nos acessos. Árbitros de barramento também são usados em monoprocessadores para controlar diferentes operações de DMA (*Direct Memory Access*). Por exemplo, no Intel Pentium a instrução LOCK informa ao árbitro de barramento que a próxima instrução de máquina deve ser atômica, com respeito ao acesso à memória. O hardware também é responsável por manter a consistência entre as *caches* dos vários processadores e a memória principal. Uma descrição mais detalhada deste tipo de arquitetura pode ser encontrada em [VAH 96].

O objetivo do sistema operacional em uma máquina SMP é oferecer a mesma visão que o usuário tem quando a máquina possui um único processador. O porte do sistema operacional de uma máquina monoprocessadora para uma máquina SMP não é exageradamente complexo, mas exige alguns cuidados. Os detalhes obviamente dependem da organização original do sistema operacional. Como diversos processadores executam simultaneamente, dois ou mais processos podem fazer chamadas de sistema ao mesmo tempo. Pensando em um *kernel* monolítico, o código do *kernel* não é reentrante, e deve ser criado um mecanismo que torne este código um recurso não compartilhável, isto é, uma seção crítica. Nesse caso, o primeiro processo adquire o "direito de executar código do *kernel*" e os demais processadores ficam esperando. Considerando a duração de algumas chamadas de sistema, esta solução é muito ineficiente. Embora seja possível imaginar melhorias neste contexto, na prática *kernel* monolítico não é usado como base para sistemas SMP. No restante dessa seção, será considerado um *kernel* convencional preemptável como organização original.

Na máquina monoprocessadora o *kernel* convencional preemptável utiliza mecanismos de sincronização clássicos para evitar que dois processos corrompam as estruturas de dados. Mecanismos de sincronização são construídos a partir de algum tipo de operação atômica. As operações atômicas mais populares no caso de monoprocessador são "desabilitar interrupções" e "instruções tipo *swap* ou *test-and-set*". Ocorre que, na presença de vários processadores, estas operações deixam de ser atômicas. Desabilitar interrupções não interrompe a execução nos demais processadores e a instrução de máquina tipo *swap* faz dois acessos a memória, o qual pode ser intercalado com outro *swap* executado em outro processador, tornando o mecanismo inútil. A solução é dotar o hardware do árbitro de barramento com a capacidade de bloquear a memória para uso exclusivo de um único processador. Se isto é feito durante a execução de uma instrução tipo "swap", fica garantida a sua atomicidade e o mecanismo de sincronização funciona. O mecanismo do *spin-lock* pode ser então usado para implementar as operações atômicas P e V dos semáforos. A partir deste momento tem-se os dois tipos fundamentais de sincronização. Logo, a chave para adaptar o *kernel* convencional para uso em máquina SMP é adaptar os mecanismos de sincronização internos do *kernel* para que funcionem também no ambiente multiprocessado.

Em geral as operações P e V dos semáforos são mais demoradas quando a seção crítica está livre. Ao mesmo tempo, o *busy-waiting* do *spin-lock* é mais eficiente quando a seção crítica, embora ocupada, será logo liberada. Como regra geral os semáforos ou qualquer outro mecanismo que bloqueia o processo são usados somente quando a espera

pode ser longa. No caso de simples exclusão mútua durante operações rápidas o *spin-lock* é preferido. Por exemplo, as próprias operações P e V são tornadas atômicas através de *spin-lock*. Boa parte dos sistemas operacionais utilizam um conjunto de mecanismos de sincronização, empregando um ou outro conforme a situação [VAH 96].

Com respeito aos deadlocks, a forma mais comum de evitá-lo dentro do *kernel* é ordenar as seções críticas e alocá-las sempre em ordem crescente, tornando impossível a ocorrência de uma lista circular [SIL 99]. Quando esta solução é impossível, pode-se utilizar construções do tipo "*try-lock*", onde o processo que tenta requisitar uma seção crítica não fica bloqueado. Neste caso é necessário prever no código alguma ação para o processo que tenta entrar em uma seção crítica mas o acesso é negado.

No momento de estabecer quais são as seções críticas do *kernel*, com o propósito de protege-las, surge a questão da granularidade. Tem-se uma proteção com **granularidade grande** quando as seções críticas correspondem a grandes trechos de código e até mesmo subsistemas completos. Por exemplo, é possível considerar todo o sistema de arquivos como uma seção crítica e controlar o acesso através de um único semáforo. Desta forma, a cada momento apenas um processo pode estar executando código do sistema de arquivos. Uma forma alternativa é usar **granularidade pequena**, protegendo cada tabela ou lista individualmente. Em geral, granularidade grande diminui o paralelismo possível dentro do *kernel*, ao passo que granularidade pequena aumenta o custo de alocar e liberar seções críticas, as quais estão livres na maior parte do tempo. A melhor solução depende do padrão de utilização do sistema operacional por parte dos processos e da probabilidade de conflitos dentro do *kernel*.

Nas máquinas SMP o escalonamento é um pouco mais complexo, em função da existência de vários processos no estado de apto. Uma característica importante é a existência de memória *cache* nos processadores. Muitas vezes é mais interessante manter um processo suspenso enquanto existe um processador livre, para que ele volte a executar no mesmo processador que executara antes, podendo-se assim aproveitar o conteúdo da *cache* de memória. Se ele for disparado em um processador diferente, será necessário gastar tempo extra para carregar a *cache* deste outro processador com as posições de memória que interessam a este processo. O mesmo pode ser dito para a TLB da MMU em algumas arquiteturas.

4.5. Estudo de caso - Linux

Linux é um sistema operacional com código fonte aberto, estilo Unix, originalmente criado por Linus Torvalds a partir de 1991 com o auxílio de desenvolvedores espalhados ao redor do mundo. Linux é "software livre" no sentido que pode ser copiado, modificado, usado de qualquer forma e redistribuído sem nenhuma restrição, sendo distribuído sob o "GNU General Public License".

O sistema operacional Linux inclui multiprogramação, memória virtual, bibliotecas compartilhadas, protocolos de rede TCP/IP e muitas outras características consistentes com um sistema multiusuário tipo Unix. Uma descrição completa do Linux não cabe neste texto. Além da página oficial http://www.linux.org, qualquer pesquisa na Internet ou na livraria vai revelar uma enorme quantidade de material sobre o assunto.

O Linux é um sistema operacional em evolução, e existem diferenças importantes entre suas versões. Este texto descreve a versão 2.2 e baseia-se, principalmente, na descrição que aparece em [BOV 01]. O Linux evolui rapidamente e alguns aspectos descritos aqui já sofreram alterações na versão 2.4. Em alguns momentos serão feitas referências à arquitetura do Pentium Intel, por ser o processador

mais usado na execução do Linux. Esta seção descreve principalmente sistemas com um único processador, mas são feitas algumas considerações sobre o ambiente SMP.

O Linux segue a linha do *kernel* convencional não-preemptável, mas inclui suporte para multiprocessamento simétrico. Dentro do *kernel* existem dois tipos principais de fluxo de controle: aqueles originados por um processo realizando uma chamada de sistema e aqueles disparados pela ocorrência de interrupções de hardware ou de proteção, estas últimas chamadas de exceções. Estes fluxos de controle internos ao *kernel* não possuem um descritor próprio e eles não são escalonados por algum algoritmo central. Muitas vezes executam do início ao fim, pois o *kernel* é não preemptável, porém a execução de vários fluxos dentro do *kernel* pode ser intercalada quando acontecem interrupções ou quando o processo solicita seu próprio bloqueio, liberando o processador voluntariamente. Este entrelaçamento da execução dentro do *kernel* exige cuidados com as estruturas de dados compartilhadas.

Um processo executando em modo *kernel* não pode ser substituído por outro processo de mais alta prioridade, a não ser quando o processo executando no *kernel* solicita explicitamente o seu próprio bloqueio, seguido de um chaveamento. O processo dentro do *kernel* pode ser interrompido pela ocorrência de interrupções de hardware e de proteção, mas ao término do respectivo tratador ele retoma a sua execução. O fluxo de controle associado com uma interrupção somente pode ser interrompido pela ocorrência de outra interrupção de hardware ou proteção. Uma conseqüência dessa organização é que chamadas de sistema não bloqueantes são atômicas com respeito às chamadas de sistema executadas por outros processos. Logo, uma estrutura de dados do *kernel* não acessada por tratadores de interrupção de hardware ou proteção não precisa ser protegida. Apenas é necessário que, antes de liberar o processador, o processo executando código do *kernel* coloque as estruturas de dados em um estado consistente. Esta é a maior vantagem da não preempção. A maior desvantagem está em fazer um processo de alta prioridade esperar que um processo de baixa prioridade termine a sua chamada de sistema antes de obter o processador.

Em alguns momentos são utilizadas "operações atômicas" para garantir a consistência de uma operação, mesmo com interrupções habilitadas. Uma operação atômica neste contexto corresponde a uma única instrução de máquina, a qual não pode ser interrompida durante sua execução. No caso do Pentium Intel, pode-se usar uma instrução do tipo lê/modifica/escreve no caso de monoprocessador, ou uma instrução do tipo lê/modifica/escreve precedida da instrução LOCK, no caso de multiprocessadores.

Muitas vezes as interrupções são simplesmente desabilitadas durante o acesso a uma estrutura de dados, para garantir que não haverá interferência de nenhum tipo. Este é o mecanismo mais freqüente no Linux. É importante observar que desabilitar interrupções não impede as interrupções de proteção (exceções). Também podem existir seções críticas aninhadas, quando as interrupções já desabilitadas são novamente desabilitadas. Neste caso, não basta habilitar as interrupções na saída da seção crítica, é necessário recolocar o valor original da flag EF encontrado no início de cada seção crítica. A seqüência fica sendo: salva as flags em uma variável, desabilita interrupções, executa a seção crítica, restaura as flags a partir da variável usada.

No Linux, um tratador de interrupção não precisa realizar todo o trabalho associado com a interrupção que ocorreu. Ele pode "anotar" parte do trabalho para ser realizado mais tarde. No Linux 2.2 este trabalho postergado é chamado de *bottom-half* e será executado em um momento conveniente para o *kernel*, tal como no término de uma chamada de sistema, no término do tratamento de uma exceção, no término do tratamento de uma interrupção ou no momento do chaveamento entre dois processos. A capacidade de postergar parte de seu trabalho é importante, pois como tratadores de

interrupção não podem ficar bloqueados como se fossem processos, eles não podem depender da disponibilidade de estruturas de dados compartilhadas para terminar seu trabalho. Existem vários tipos de *bottom-half*, e o momento de execução de cada um depende dos recursos que eles necessitam. O mecanismo do *bottom-half* é um passo na direção de *threads* de *kernel*, as quais representam uma solução mais elegante.

Em alguns momentos o código do *kernel* utiliza semáforos para proteger seções críticas. Os semáforos são definidos da forma tradicional, como uma estrutura de dados composta por um contador inteiro, uma lista de processos bloqueados e mais campos auxiliares. Na implementação das operações P e V, as interrupções são desabilitadas para garantir sua atomicidade em monoprocessadores. Em alguns momentos dentro do *kernel* o fluxo de controle não pode ficar bloqueado, e neste caso é utilizada uma variação da operação P que, em caso do recurso ocupado, retorna um código de erro mas não causa o bloqueio. Também existe uma variação da operação P usada em *device-drivers*, na qual o processo pode ficar bloqueado mas continua sensível aos sinais Unix, os quais podem fazer este processo acordar e desistir de obter o semáforo.

Como o *kernel* não é preemptável, poucos semáforos são utilizados. No caso de monoprocessamento, semáforos são usados apenas quando o processo poderá ficar bloqueado mais tarde em função de um acesso a disco, ou quando um tratador de interrupção pode acessar uma estrutura de dados global do *kernel*. Eles são encontrados principalmente na gerência de memória e no sistema de arquivos. Deadlock é evitado através da ordenação dos semáforos e alocação deles em uma certa ordem [SIL 99].

Em máquinas SMP diversos processadores podem executar em modo supervisor simultaneamente, logo os benefícios de um *kernel* não preemptável são reduzidos. É necessária uma sincronização explícita nas seções críticas do *kernel* através de semáforos e/ou *spin-locks*. Linux versão 2.0 usava uma solução bem simples: A qualquer instante, no máximo um processador podia acessar as estruturas de dados do *kernel* e tratar interrupções. Esta solução limitava o paralelismo no sistema.

Na versão 2.2 muitas seções críticas foram tratadas individualmente e a restrição geral foi removida. *Spin-locks* são usados. A solução implica em *busy-waiting*, pois o processador continua a executar o laço do *spin-lock* enquanto espera a seção crítica ser liberada. Se a seção crítica for rápida, ainda é melhor isto do que realizar um chaveamento de processo. Observe que o *spin-lock* impede o acesso à estrutura de dados por outro processador, mas não impede que tratadores de interrupção disparados no mesmo processador que obteve o *spin-lock* acessem estas estruturas. Logo, são usadas macros que, ao mesmo tempo, adquirem o *spin-lock* e desabilitam as interrupções. Além do *spin-lock* simples (libera ou proibe qualquer tipo de acesso), o *kernel* do Linux utiliza uma variação chamada "*read/write spin-lock*", o qual permite o acesso simultâneo de vários processos leitores, mas garante exclusão mútua na escrita.

A versão 2.2 ainda inclui um *spin-lock* global do *kernel*, usado para proteger coletivamente todas as estruturas de dados usadas no acesso a arquivos, a maioria das relacionadas com comunicação via rede, todas as relacionadas com IPC (*Inter-Process Communication*) entre processos de usuário e ainda outras mais. Qualquer rotina que acesse uma dessas estruturas de dados precisa obter este *spin-lock* antes, o que significa que um grande número de chamadas de sistema no Linux não podem executar simultaneamente em diferentes processadores. Fosse o Linux um *kernel* convencional preemptável, o paralelismo potencial do SMP seria muito melhor aproveitado. Mesmo assim, várias estruturas de dados já estão protegidas por *spin-locks* individuais. A granularidade menor no mecanismo de sincronização proporciona maior paralelismo e desempenho. Entretanto, sua inclusão em um *kernel* originalmente monoprocessado e não preemptivo deve ser muito cuidadosa para evitar *deadlock* e outras anomalias.

É importante destacar que o *kernel* do Linux está em constante evolução. Por exemplo, já na versão 2.4 o mecanismo de *bottom-half* foi melhorado de tal forma que possa existir execução simultânea destes blocos de código em diferentes processadores, aumentando o desempenho do sistema. Em geral, é possível dizer que a evolução do *kernel* do Linux o leva, a cada passo, mais perto da idéia de um programa concorrente onde processos executam concorrentemente no *kernel* e podem ser preemptados a qualquer momento. A longo prazo o *kernel* do Linux deverá tornar-se mais um programa concorrente típico e menos um aglomerado de tratamentos diferenciados para dezenas de situações específicas. A descrição do *kernel* do Linux feita aqui foi simplificada. Em [BOV 01] são dedicadas mais de 600 páginas ao assunto.

4.6. Conclusões

Ao longo dos últimos 40 anos, muito esforço foi empreendido no sentido de criar as estruturas de software mais apropriadas para organizar internamente os sistemas operacionais. Existem diferenças entre os serviços oferecidos por um pequeno sistema operacional de propósito específico e um sistema operacional de propósito geral. Também existem diferenças entre o ambiente computacional da década de 60 e este do início do século 21. Apesar disto, o número de diferentes organizações para sistemas operacionais não é grande. Também é razoável afirmar que, desde o início dos anos 70, o sistema operacional Unix exerceu indiscutível influência sobre os projetistas da área.

Este texto discutiu as possíveis organizações para um *kernel* (núcleo) de sistema operacional. Inicialmente foi feita uma rápida revisão dos conceitos fundamentais dos sistemas operacionais e dos mecanismos clássicos para a sincronização de processos. Em seguida, diversas organizações possíveis para o *kernel* do sistema operacional foram apresentadas. Em muitas delas o *kernel* apareceu como um programa concorrente, onde a preocupação com a sincronização entre processos está presente. O texto buscou definir uma taxonomia para orientar o leitor no entendimento das organizações existentes e nas conseqüências de cada uma delas. O Linux foi usado como estudo de caso.

O sistema operacional sempre teve o papel de um administrador, tentando compatibilizar, da melhor forma possível, as demandas dos programas de usuário com os recursos existentes no hardware. Atualmente estão acontecendo importante mudanças nestas duas áreas. Os recursos disponíveis no hardware continuam a aumentar a taxas geométricas, sendo a capacidade dos discos, a velocidade das redes, o tamanho da memória principal e a interconectividade as grandes vedetes. Por outro lado, aplicações distribuídas e/ou envolvendo mídias contínuas são cada vez mais requisitadas e problemas com segurança aumentam a cada dia. Também estão sendo dissiminadas plataformas do tipo *wireless* e *embedded*, com demandas específicas e recursos limitados. Após várias décadas de lenta evolução, existe o sentimento de que os sistemas operacionais deverão em breve necessitar um salto de flexibilidade em sua organização, para atender as demandas da computação no século 21. Com certeza, as organizações apresentadas neste texto continuarão a evoluir nos próximos anos.

4.7. Agradecimentos

Os autores agradecem a Carlos Montez, Fábio Rodrigues de la Rocha, Mauro M. Mattos e Rafael R. Obelheiro, pelas várias sugestões e comentários.

4.8. Bibliografia

[BOE 81] BOEHM, B. **Software Engineering Economics**. Prentice-Hall, 1981.

[BOV 01] BOVET, D.P.; CESATI, M. **Understanding the Linux Kernel**. O'Reilly & Associates, 2001. 684p.

[DIJ 68] DIJKSTRA, E.W. **The Structure of the THE Multiprogramming System**. Comm. of the ACM, vol. 11, n. 5, pp. 341-346, May 1968.

[HOL 83] HOLT, R.C. **Concurrent Euclid, The Unix System, and Tunis**. Addison-Wesley, 1983. 323p.

[OLI 01] OLIVEIRA, R.S. de; CARISSIMI, A. da S.; TOSCANI, S.S. **Sistemas Operacionais**. Porto Alegre: Sagra-Luzzatto, 2001. 233p. {Série Livros Didáticos do Instituto de Informática da UFRGS, N.11}

[PRE 01] PRESSMAN, R.S. **Software Engineering: A Practitioner's Approach**. 5th edition. McGraw-Hill Higher Education, 2001. 860p.

[SIL 99] SILBERSCHATZ, A.; GALVIN, P.B. **Operating Systems Concepts**. 5th edition. John Wiley & Sons, 1999. 888p.

[STA 01] STALLINGS, W. **Operating Systems**. 4th edition. Prentice-Hall, 2001. 779p.

[TAN 95] TANENBAUM, A.S. **Distributed Operating Systems.** Prentice-Hall, 1995. 614p.

[TAN 00] TANENBAUM, A.S.; WOODHULL, A.S. **Sistemas Operacionais: Projeto e Implementação**. 2ª edição. Bookman, 2000. 759p.

[VAH 96] VAHALIA, U. **Unix Internals: The New Frontiers**. Prentice-Hall, 1996. 601p.