Sistemas Operacionais
I - Conceitos Básicos [*†]

Prof. Carlos Alberto Maziero
PPGIa CCET PUCPR
http://www.ppgia.pucpr.br/~maziero

30 de julho de 2008

Resumo

Um sistema de computação é constituído basicamente por hardware e software. O hardware é composto por circuitos eletrônicos (processador, memória, portas de entrada/saída, etc) e periféricos eletro-óptico-mecânicos (teclados, mouses, discos rígidos, unidades de disquete, CD ou DVD, dispositivos USB, etc). Por sua vez, o software de aplicação é representado por programas destinados ao usuário do sistema, que constituem a razão final de seu uso, como editores de texto, navegadores Internet ou jogos. Entre os aplicativos e o hardware reside uma camada de software multi-facetada e complexa, denominada genericamente de *Sistema Operacional*. Neste capítulo veremos quais os objetivos básicos do sistema operacional, quais desafios ele deve resolver e como ele é estruturado para alcançar seus objetivos.

[†]Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2ε, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criador de gráficos *GNUPlot* e processador PS/PDF *GhostScript*, entre outros.

Sumário

1 Objetivos

Existe uma grande distância entre os circuitos eletrônicos e dispositivos de hardware e os programas aplicativos em software. Os circuitos são complexos, acessados através de interfaces de baixo nível (geralmente usando as portas de entrada/saída do processador) e muitas vezes suas características e seu comportamento dependem da tecnologia usada em sua construção. Por exemplo, a forma de acesso de baixo nível a discos rígidos IDE difere da forma de acesso a discos SCSI ou leitores de CD. Essa grande diversidade pode ser uma fonte de dores de cabeça para o desenvolvedor de aplicativos. Portanto, torna-se desejável oferecer aos programas aplicativos uma forma de acesso homogênea aos dispositivos físicos, que permita abstrair as diferenças tecnológicas entre eles.

O sistema operacional é uma camada de software que opera entre o hardware e os programas aplicativos voltados ao usuário final. O sistema operacional é uma estrutura de software ampla, muitas vezes complexa, que incorpora aspectos de baixo nível (como drivers de dispositivos e gerência de memória física) e de alto nível (como programas utilitários e a própria interface gráfica).

A figura 1 ilustra a arquitetura geral de um sistema de computação típico. Nela, podemos observar elementos de hardware, o sistema operacional e alguns programas aplicativos.

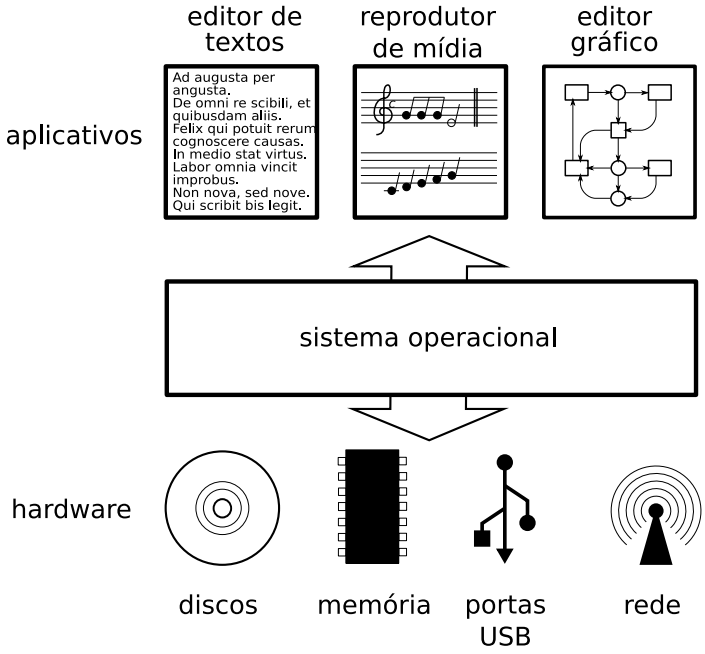

Figura 1: Estrutura de um sistema de computação típico

Os objetivos básicos de um sistema operacional podem ser sintetizados em duas palavras-chave: "abstração" e "gerência", cujos principais aspectos são detalhados a seguir.

1.1 Abstração de recursos

Acessar os recursos de hardware de um sistema de computação pode ser uma tarefa complexa, devido às características específicas de cada dispositivo físico e a complexidade de suas interfaces. Por exemplo, a seqüência a seguir apresenta os principais passos envolvidos na abertura de um arquivo (operação `open`) em um leitor de disquete:

1. verificar se os parâmetros informados estão corretos (nome do arquivo, identificador do leitor de disquete, buffer de leitura, etc);
2. verificar se o leitor de disquetes está disponível;
3. verificar se o leitor contém um disquete;
4. ligar o motor do leitor e aguardar atingir a velocidade de rotação correta;
5. posicionar a cabeça de leitura sobre a trilha onde está a tabela de diretório;
6. ler a tabela de diretório e localizar o arquivo ou subdiretório desejado;
7. mover a cabeça de leitura para a posição do bloco inicial do arquivo;
8. ler o bloco inicial do arquivo e depositá-lo em um buffer de memória.

Assim, o sistema operacional deve definir interfaces abstratas para os recursos do hardware, visando atender os seguintes objetivos:

- *Prover interfaces de acesso aos dispositivos, mais simples de usar que as interface de baixo nível*, para simplificar a construção de programas aplicativos. Por exemplo: para ler dados de um disco rígido, uma aplicação usa um conceito chamado *arquivo*, que implementa uma visão abstrata do disco rígido, acessível através de operações como `open`, `read` e `close`. Caso tivesse de acessar o disco diretamente, teria de manipular portas de entrada/saída e registradores com comandos para o controlador de disco (sem falar na dificuldade de localizar os dados desejados dentro do disco).
- *Tornar os aplicativos independentes do hardware*. Ao definir uma interface abstrata de acesso a um dispositivo de hardware, o sistema operacional desacopla o hardware dos aplicativos e permite que ambos evoluam de forma mais autônoma. Por exemplo, o código de um editor de textos não deve ser dependente da tecnologia de discos rígidos utilizada no sistema.
- *Definir interfaces de acesso homogêneas para dispositivos com tecnologias distintas*. Através de suas abstrações, o sistema operacional permite aos aplicativos usar a mesma interface para dispositivos diversos. Por exemplo, um aplicativo acessa dados em disco através de arquivos e diretórios, sem precisar se preocupar com a estrutura real de armazenamento dos dados, que podem estar em um disquete, um disco IDE, uma máquina fotográfica digital conectada à porta USB, um CD ou mesmo um disco remoto, compartilhado através da rede.

1.2 Gerência de recursos

Os programas aplicativos usam o hardware para atingir seus objetivos: ler e armazenar dados, editar e imprimir documentos, navegar na Internet, tocar música, etc. Em um sistema com várias atividades simultâneas, podem surgir conflitos no uso do hardware, quando dois ou mais aplicativos precisam dos mesmos recursos para poder executar. Cabe ao sistema operacional definir *políticas* para gerenciar o uso dos recursos de hardware pelos aplicativos, e resolver eventuais disputas e conflitos. Vejamos algumas situações onde a gerência de recursos do hardware se faz necessária:

- Cada computador possui normalmente um só processador. O uso desse processador deve ser distribuído entre os aplicativos presentes no sistema, de forma que cada um deles possa executar na velocidade adequada para cumprir suas funções sem prejudicar os outros. O mesmo ocorre com a memória RAM, que deve ser distribuída de forma justa entre as aplicações.

- A impressora é um recurso cujo acesso deve ser efetuado de forma mutuamente exclusiva (apenas um aplicativo por vez), para não ocorrer mistura de conteúdo nos documentos impressos. O sistema operacional resolve essa questão definindo uma fila de trabalhos a imprimir (*print jobs*) normalmente atendidos de forma seqüencial (FIFO).

- Ataques de negação de serviço (*DoS – Denial of Service*) são comuns na Internet. Eles consistem em usar diversas técnicas para forçar um servidor de rede a dedicar seus recursos a atender um determinado usuário, em detrimento dos demais. Por exemplo, ao abrir milhares conexões simultâneas em um servidor de e-mail, um atacante pode reservar para si todos os recursos do servidor (processos, conexões de rede, memória e processador), fazendo com que os demais usuários não sejam mais atendidos. É responsabilidade do sistema operacional do servidor detectar tais situações e impedir que todos os recursos do sistema sejam monopolizados por um só usuário (ou um pequeno grupo).

Assim, um sistema operacional visa abstrair o acesso e gerenciar os recursos de hardware, provendo aos aplicativos um ambiente de execução abstrato, no qual o acesso aos recursos se faz através de interfaces simples, independentes das características e detalhes de baixo nível, e no qual os conflitos no uso do hardware são minimizados.

2 Tipos de sistemas operacionais

Os sistemas operacionais podem ser classificados segundo diversos parâmetros e perspectivas, como tamanho, velocidade, suporte a recursos específicos, acesso à rede, etc. A seguir são apresentados alguns tipos de sistemas operacionais usuais (muitos sistemas operacionais se encaixam bem em mais de uma das categorias apresentadas):

Batch (de lote) : os sistemas operacionais mais antigos trabalhavam "por lote", ou seja, todos os programas a executar eram colocados em uma fila, com seus dados e demais informações para a execução. O processador recebia um programa após o outro, processando-os em seqüência, o que permitia um alto grau de utilização do sistema. Ainda hoje o termo "em lote" é usado para designar um conjunto de comandos que deve ser executado em seqüência, sem interferência do usuário. Exemplos desses sistemas incluem o OS/360 e VMS, entre outros.

De rede : um sistema operacional de rede deve possuir suporte à operação em rede, ou seja, a capacidade de oferecer às aplicações locais recursos que estejam localizados em outros computadores da rede, como arquivos e impressoras. Ele também deve disponibilizar seus recursos locais aos demais computadores, de forma controlada. A maioria dos sistemas operacionais atuais oferece esse tipo de funcionalidade.

Distribuído : em um sistema operacional distribuído, os recursos de cada máquina estão disponíveis globalmente, de forma transparente aos usuários. Ao lançar uma aplicação, o usuário interage com sua janela, mas não sabe onde ela está executando ou armazenando seus arquivos: o sistema é quem decide, de forma transparente. Os sistemas operacionais distribuídos já existem há tempos (Amoeba [Tanenbaum et al., 1991] e Clouds [Dasgupta et al., 1991], por exemplo), mas ainda não são uma realidade de mercado.

Multi-usuário : um sistema operacional multi-usuário deve suportar a identificação do "dono" de cada recurso dentro do sistema (arquivos, processos, áreas de memória, conexões de rede) e impor regras de controle de acesso para impedir o uso desses recursos por usuários não autorizados. Essa funcionalidade é fundamental para a segurança dos sistemas operacionais de rede e distribuídos. Grande parte dos sistemas atuais são multi-usuários.

Desktop : um sistema operacional "de mesa" é voltado ao atendimento do usuário doméstico e corporativo para a realização de atividades corriqueiras, como edição de textos e gráficos, navegação na Internet e reprodução de mídias simples. Sua principais características são a interface gráfica, o suporte à interatividade e a operação em rede. Exemplos de sistemas *desktop* são o Windows XP, MacOS X e Linux.

Servidor : um sistema operacional servidor deve permitir a gestão eficiente de grandes quantidades de recursos (disco, memória, processadores), impondo prioridades e limites sobre o uso dos recursos pelos usuários e seus aplicativos. Normalmente um sistema operacional servidor também tem suporte a rede e multi-usuários.

Embutido : um sistema operacional é dito embutido (*embedded*) quando é construído para operar sobre um hardware com poucos recursos de processamento, armazenamento e energia. Aplicações típicas desse tipo de sistema aparecem em telefones celulares, controladores industriais e automotivos, equipamentos eletrônicos

de uso doméstico (leitores de DVD, TVs, fornos-micro-ondas, centrais de alarme, etc.). Muitas vezes um sistema operacional embutido se apresenta na forma de uma biblioteca a ser ligada ao programa da aplicação (que é fixa). Exemplos de sistemas operacionais embutidos são o μC/OS, Xylinx, LynxOS e VxWorks.

Tempo real : ao contrário da concepção usual, um sistema operacional de tempo real não precisa ser necessariamente ultra-rápido; sua característica essencial é ter um comportamento temporal previsível (ou seja, seu tempo de resposta deve ser conhecido no melhor e pior caso de operação). A estrutura interna de um sistema operacional de tempo real deve ser construída de forma a minimizar esperas e latências imprevisíveis, como tempos de acesso a disco e sincronizações excessivas.

Existem duas classificações de sistemas de tempo real: *soft real-time systems*, nos quais a perda de prazos implica na degradação do serviço prestado. Um exemplo seria o suporte à gravação de CDs ou à reprodução de músicas. Caso o sistema se atrase, pode ocorrer a perda da mídia em gravação ou falhas na música que está sendo tocada. Por outro lado, nos *hard real-time systems* a perda de prazos pelo sistema pode perturbar o objeto controlado, com graves conseqüências humanas, econômicas ou ambientais. Exemplos desse tipo de sistema seriam o controle de funcionamento de uma turbina de avião a jato ou de uma caldeira industrial.

Exemplos de sistemas de tempo real incluem o QNX, RT-Linux e VxWorks. Muitos sistemas embutidos têm características de tempo real, e vice-versa.

3 Funcionalidades

Para cumprir seus objetivos de abstração e gerência, o sistema operacional deve atuar em várias frentes. Cada um dos recursos do sistema possui suas particularidades, o que impõe exigências específicas para gerenciar e abstrair os mesmos. Sob esta perspectiva, as principais funcionalidades implementadas por um sistema operacional típico são:

Gerência do processador : também conhecida como gerência de processos ou de atividades, esta funcionalidade visa distribuir a capacidade de processamento de forma justa[1] entre as aplicações, evitando que uma aplicação monopolize esse recurso e respeitando as prioridades dos usuários. O sistema operacional provê a ilusão de que existe um processador independente para cada tarefa, o que facilita o trabalho dos programadores de aplicações e permite a construção de sistemas mais interativos. Também faz parte da gerência de atividades fornecer abstrações para sincronizar atividades inter-dependentes e prover formas de comunicação entre elas.

[1] Distribuir de forma justa, mas não necessariamente igual, pois as aplicações têm demandas de processamento distintas; por exemplo, um navegador de Internet demanda menos o processador que um aplicativo de edição de vídeo, e por isso o navegador pode receber menos tempo de processador.

Gerência de memória : tem como objetivo fornecer a cada aplicação uma área de memória própria, independente e isolada das demais aplicações e inclusive do núcleo do sistema. O isolamento das áreas de memória das aplicações melhora a estabilidade e segurança do sistema como um todo, pois impede aplicações com erros (ou aplicações maliciosas) de interferir no funcionamento das demais aplicações. Além disso, caso a memória RAM existente seja insuficiente para as aplicações, o sistema operacional pode aumentá-la de forma transparente às aplicações, usando o espaço disponível em um meio de armazenamento secundário (como um disco rígido). Uma importante abstração construída pela gerência de memória é a noção de *memória virtual*, que desvincula os endereços de memória vistos por cada aplicação dos endereços acessados pelo processador na memória RAM. Com isso, uma aplicação pode ser carregada em qualquer posição livre da memória, sem que seu programador tenha de se preocupar com os endereços de memória onde ela irá executar.

Gerência de dispositivos : cada periférico do computador possui suas peculiaridades; assim, o procedimento de interação com uma placa de rede é completamente diferente da interação com um disco rígido SCSI. Todavia, existem muitos problemas e abordagens em comum para o acesso aos periféricos. Por exemplo, é possível criar uma abstração única para a maioria dos dispositivos de armazenamento como *pen-drives*, discos SCSI ou IDE, disquetes, etc, na forma de um vetor de blocos de dados. A função da gerência de dispositivos (também conhecida como *gerência de entrada/saída*) é implementar a interação com cada dispositivo por meio de *drivers* e criar modelos abstratos que permitam agrupar vários dispositivos distintos sob a mesma interface de acesso.

Gerência de arquivos : esta funcionalidade é construída sobre a gerência de dispositivos e visa criar arquivos e diretórios, definindo sua interface de acesso e as regras para seu uso. É importante observar que os conceitos abstratos de arquivo e diretório são tão importantes e difundidos que muitos sistemas operacionais os usam para permitir o acesso a recursos que nada tem a ver com armazenamento. Exemplos disso são as conexões de rede (nos sistemas UNIX e Windows, cada socket TCP é visto como um descritor de arquivo no qual pode-se ler ou escrever dados) e as informações do núcleo do sistema (como o diretório `/proc` do UNIX). No sistema operacional experimental *Plan 9* [Pike et al., 1993], todos os recursos do sistema operacional são vistos como arquivos.

Gerência de proteção : com computadores conectados em rede e compartilhados por vários usuários, é importante definir claramente os recursos que cada usuário pode acessar, as formas de acesso permitidas (leitura, escrita, etc) e garantir que essas definições sejam cumpridas. Para proteger os recursos do sistema contra acessos indevidos, é necessário: a) definir usuários e grupos de usuários; b) identificar os usuários que se conectam ao sistema, através de procedimentos de autenticação; c) definir e aplicar regras de controle de acesso aos recursos, relacionando todos os

usuários, recursos e formas de acesso e aplicando essas regras através de procedimentos de autorização; e finalmente d) registrar o uso dos recursos pelos usuários, para fins de auditoria e contabilização.

Além dessas funcionalidades básicas oferecidas pela maioria dos sistemas operacionais, várias outras vêm se agregar aos sistemas modernos, para cobrir aspectos complementares, como a interface gráfica, suporte de rede, fluxos multimídia, gerência de energia, etc.

As funcionalidades do sistema operacional geralmente são inter-dependentes: por exemplo, a gerência do processador depende de aspectos da gerência de memória, assim como a gerência de memória depende da gerência de dispositivos e da gerência de proteção. Alguns autores [Silberschatz et al., 2001, Tanenbaum, 2003] representam a estrutura do sistema operacional conforme indicado na figura 2. Nela, o núcleo central implementa o acesso de baixo nível ao hardware, enquanto os módulos externos representam as várias funcionalidades do sistema.

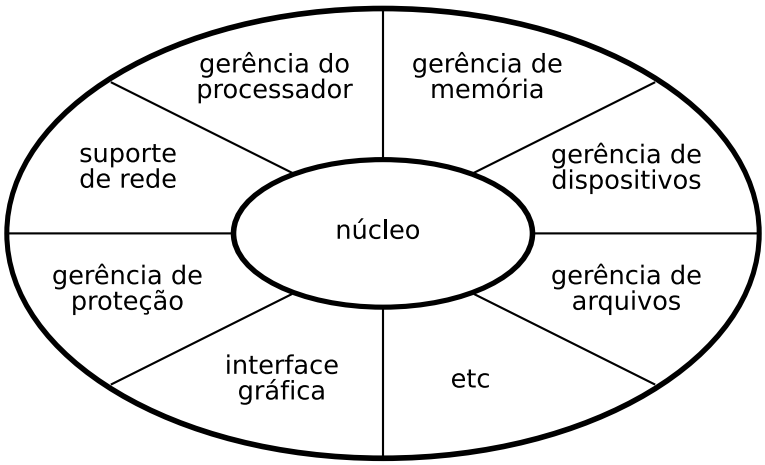

Figura 2: Funcionalidades do sistema operacional

Uma regra importante a ser observada na construção de um sistema operacional[2] é a separação entre os conceitos de política e mecanismo. Como *política* consideram-se os aspectos de decisão mais abstratos, que podem ser resolvidos por algoritmos de nível mais alto, como por exemplo decidir a quantidade de memória que cada aplicação ativa deve receber, ou qual o próximo pacote de rede a enviar para satisfazer determinadas especificações de qualidade de serviço.

Por outro lado, como *mecanismo* consideram-se os procedimentos de baixo nível usados para implementar as políticas, ou seja, atribuir ou retirar memória de uma aplicação, enviar ou receber um pacote de rede, etc. Os mecanismos devem ser suficientemente genéricos para suportar mudanças de política sem necessidade de modificações. Essa

[2]Na verdade essa regra é tão importante que deveria ser levada em conta na construção de qualquer sistema complexo.

separação entre os conceitos de política e mecanismo traz uma grande flexibilidade aos sistemas operacionais, permitindo alterar sua personalidade (sistemas mais interativos ou mais eficientes) sem ter de alterar o código que interage diretamente com o hardware. Alguns sistemas, como o InfoKernel [Arpaci-Dusseau et al., 2003], permitem que as aplicações escolham as políticas do sistema mais adequadas às suas necessidades.

4 Estrutura de um sistema operacional

Um sistema operacional não é um bloco único e fechado de software executando sobre o hardware. Na verdade, ele é composto de diversos componentes com objetivos e funcionalidades complementares. Alguns dos componentes mais relevantes de um sistema operacional típico são:

Núcleo : é o coração do sistema operacional, responsável pela gerência dos recursos do hardware usados pelas aplicações. Ele também implementa as principais abstrações utilizadas pelos programas aplicativos.

Drivers : módulos de código específicos para acessar os dispositivos físicos. Existe um driver para cada tipo de dispositivo, como discos rígidos IDE, SCSI, portas USB, placas de vídeo, etc. Muitas vezes o driver é construído pelo próprio fabricante do hardware e fornecido em forma compilada (em linguagem de máquina) para ser acoplado ao restante do sistema operacional.

Código de inicialização : a inicialização do hardware requer uma série de tarefas complexas, como reconhecer os dispositivos instalados, testá-los e configurá-los adequadamente para seu uso posterior. Outra tarefa importante é carregar o núcleo do sistema operacional em memória e iniciar sua execução.

Programas utilitários : são programas que facilitam o uso do sistema computacional, fornecendo funcionalidades complementares ao núcleo, como formatação de discos e mídias, configuração de dispositivos, manipulação de arquivos (mover, copiar, apagar), interpretador de comandos, terminal, interface gráfica, gerência de janelas, etc.

As diversas partes do sistema operacional se relacionam entre si conforme apresentado na figura 3. A forma como esses diversos componentes são interligados e se relacionam varia de sistema para sistema; algumas possibilidades são discutidas na seção 6.

5 Conceitos de hardware

O sistema operacional interage diretamente com o hardware para fornecer serviços às aplicações. Para a compreensão dos conceitos implementados pelos sistemas operacionais, é necessário ter uma visão clara dos recursos fornecidos pelo hardware e a forma

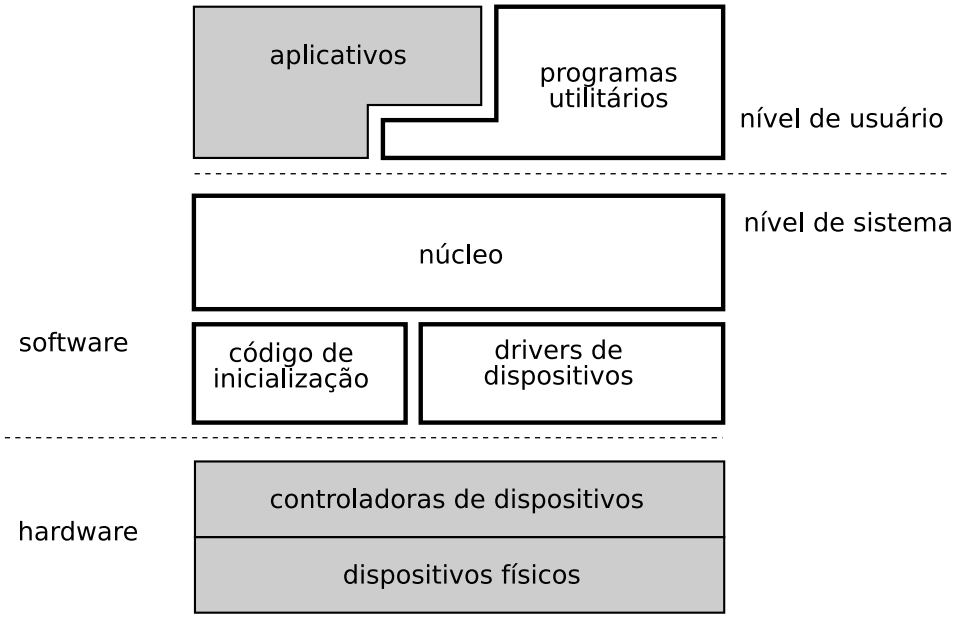

Figura 3: Estrutura de um sistema operacional

de acessá-los. Esta seção apresenta uma revisão dos principais aspectos do hardware de um computador pessoal convencional.

Um sistema de computação típico é constituído de um ou mais processadores, responsáveis pela execução das instruções das aplicações, uma área de memória que armazena as aplicações em execução (seus códigos e dados) e dispositivos periféricos que permitem o armazenamento de dados e a comunicação com o mundo exterior, como discos rígidos, terminais e teclados. A maioria dos computadores mono-processados atuais segue uma arquitetura básica definida nos anos 40 por János (John) Von Neumann, conhecida por "arquitetura Von Neumann". A principal característica desse modelo é a idéia de "programa armazenado", ou seja, o programa a ser executado reside na memória junto com os dados. Os principais elementos constituintes do computador estão interligados por um ou mais barramentos (para a transferência de dados, endereços e sinais de controle). A figura 4 ilustra a arquitetura de um computador típico.

O núcleo do sistema de computação é o processador. Ele é responsável por continuamente ler instruções e dados da memória ou de periféricos, processá-los e enviar os resultados de volta à memória ou a outros periféricos. Um processador convencional é normalmente constituído de uma unidade lógica e aritmética (ULA), que realiza os cálculos e operações lógicas, um conjunto de registradores para armazenar dados de trabalho e alguns registradores para funções especiais (contador de programa, ponteiro de pilha, flags de status, etc).

Todas as transferência de dados entre processador, memória e periféricos são feitas através dos barramentos: o **barramento de endereços** indica a posição de memória (ou o dispositivo) a acessar, o **barramento de controle** indica a operação a efetuar (leitura ou

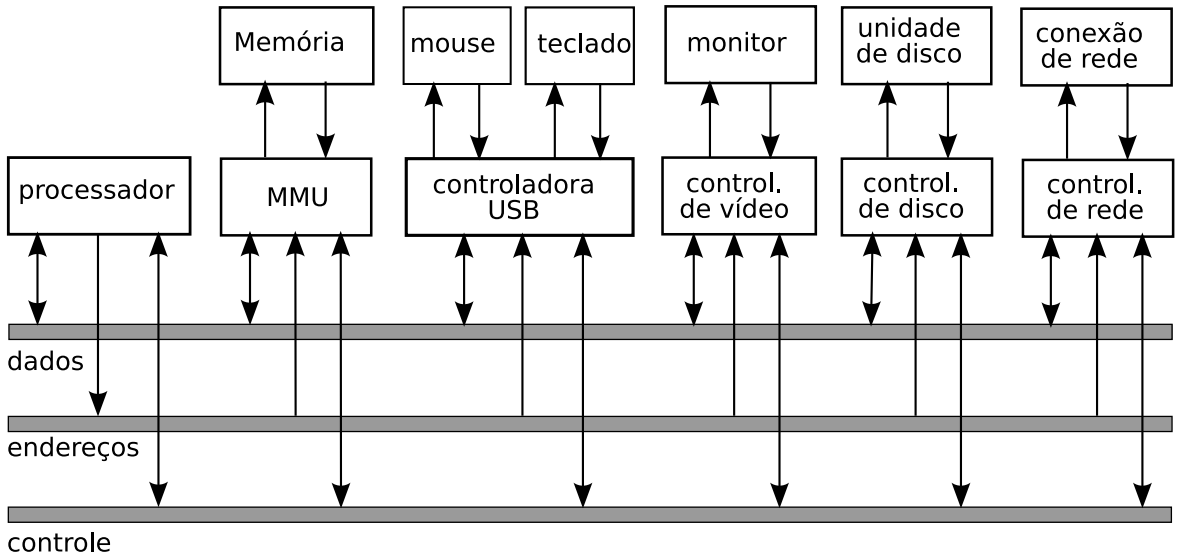

Figura 4: Arquitetura de um computador típico

escrita) e o **barramento de dados** transporta a informação indicada entre o processador e a memória ou um controlador de dispositivo.

O acesso à memória é geralmente mediado por um controlador específico (que pode estar fisicamente dentro do próprio processador): a **Unidade de Gerência de Memória** (MMU - *Memory Management Unit*). Ela é responsável por analisar cada endereço solicitado pelo processador, validá-los, efetuar as conversões de endereçamento necessárias e executar a operação solicitada pelo processador (leitura ou escrita de uma posição de memória).

Os periféricos do computador (discos, teclado, monitor, etc) são acessados através de circuitos específicos genericamente denominados **controladores**: a placa de vídeo permite o acesso ao monitor, a placa ethernet dá acesso à rede, o controlador USB permite acesso ao mouse, teclado e outros dispositivos USB externos. Para o processador, cada dispositivo é representado por seu respectivo controlador. Os controladores podem ser acessados através de *portas de entrada/saída* endereçáveis: a cada controlador é atribuída uma faixa de endereços de portas de entrada/saída. A tabela 1 a seguir apresenta alguns endereços portas de entrada/saída para acessar controladores em um PC típico:

5.1 Interrupções

Quando um controlador de periférico tem uma informação importante a fornecer ao processador, ele tem duas alternativas de comunicação:

- Aguardar até que o processador o consulte, o que poderá ser demorado caso o processador esteja ocupado com outras tarefas (o que geralmente ocorre);

dispositivo	**endereços de acesso**
teclado	0060h-006Fh
barramento IDE primário	0170h-0177h
barramento IDE secundário	01F0h-01F7Fh
porta serial COM1	02F8h-02FFh
porta serial COM2	03F8h-03FFh

Tabela 1: Endereços de acesso a dispositivos

- Notificar o processador através do barramento de controle, enviando a ele uma *requisição de interrupção* (IRQ – *Interrupt ReQuest*).

Ao receber a requisição de interrupção, os circuitos do processador suspendem seu fluxo de execução corrente e desviam para um endereço pré-definido, onde se encontra uma *rotina de tratamento de interrupção* (*interrupt handler*). Essa rotina é responsável por tratar a interrupção, ou seja, executar as ações necessárias para atender o dispositivo que a gerou. Ao final da rotina de tratamento da interrupção, o processador retoma o código que estava executando quando recebeu a requisição.

A figura 5 representa os principais passos associados ao tratamento de uma interrupção envolvendo a placa de rede Ethernet, enumerados a seguir:

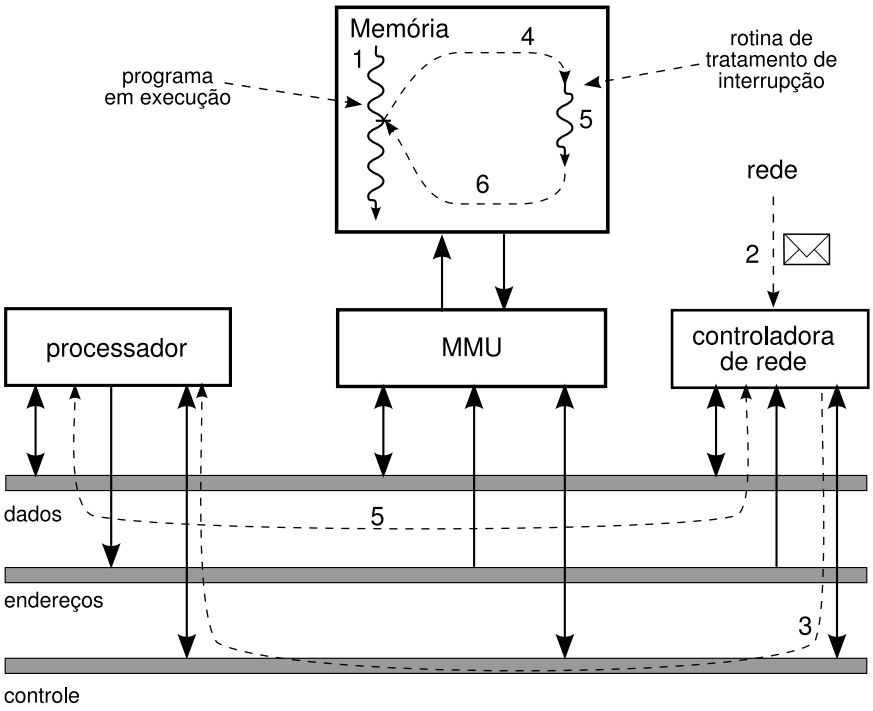

Figura 5: Roteiro típico de um tratamento de interrupção

1. O processador está executando um programa qualquer (em outras palavras, um fluxo de execução);
2. Um pacote vindo da rede é recebido pela placa Ethernet;
3. A placa envia uma solicitação de interrupção (IRQ) ao processador;
4. O processamento é desviado do programa em execução para a rotina de tratamento da interrupção;
5. A rotina de tratamento é executada para receber as informações da placa de rede (via barramentos de dados e de endereços) e atualizar as estruturas de dados do sistema operacional;
6. A rotina de tratamento da interrupção é finalizada e o processador retorna à execução do programa que havia sido interrompido.

Esse roteiro de ações ocorre a cada requisição de interrupção recebida pelo processador. Cada interrupção geralmente corresponde a um evento ocorrido em um dispositivo periférico: a chegada de um pacote de rede, um click no mouse, uma operação concluída pelo controlador de disco, etc. Isso representa centenas ou mesmo milhares de interrupções recebidas por segundo, dependendo da carga e da configuração do sistema (número e natureza dos periféricos). Por isso, as rotinas de tratamento de interrupção devem ser curtas e realizar suas tarefas rapidamente (para não prejudicar o desempenho do sistema).

Normalmente o processador recebe e trata cada interrupção recebida, mas nem sempre isso é possível. Por exemplo, receber e tratar uma interrupção pode ser problemático caso o processador já esteja tratando outra interrupção. Por essa razão, o processador pode decidir ignorar temporariamente algumas interrupções, se necessário. Isso é feito ajustando o bit correspondente à interrupção em um registrador específico do processador.

Para distinguir interrupções geradas por dispositivos distintos, cada interrupção é identificada por um inteiro, normalmente com 8 bits. Como cada interrupção pode exigir um tipo de tratamento diferente (pois os dispositivos são diferentes), cada IRQ deve disparar sua própria rotina de tratamento de interrupção. A maioria das arquiteturas atuais define um vetor de endereços de funções denominado *Vetor de Interrupções* (IV - *Interrupt Vector*); cada entrada desse vetor aponta para a rotina de tratamento da interrupção correspondente. Por exemplo, se a entrada 5 do vetor contém o valor `3C20h`, então a rotina de tratamento da IRQ 5 iniciará na posição `3C20h` da memória RAM. O vetor de interrupções reside em uma posição fixa da memória RAM, definida pelo fabricante do processador, ou tem sua posição indicada pelo conteúdo de um registrador da CPU específico para esse fim.

As interrupções recebidas pelo processador têm como origem eventos externos a ele, ocorridos nos dispositivos periféricos e reportados por seus controladores. Entretanto, alguns eventos gerados pelo próprio processador podem ocasionar o desvio da execução

usando o mesmo mecanismo das interrupções: são as *exceções*. Eventos como instruções ilegais (inexistentes ou com operandos inválidos), tentativa de divisão por zero ou outros erros de software disparam exceções no processador, que resultam na ativação de uma rotina de tratamento de exceção, usando o mesmo mecanismo das interrupções (e o mesmo vetor de endereços de funções). A tabela 2 representa o vetor de interrupções do processador Intel Pentium (extraída de [Patterson and Henessy, 2005]).

Tabela 2: Vetor de Interrupções do processador Pentium [Patterson and Henessy, 2005]

IRQ	**Descrição**
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	Intel reserved
16	floating point error
17	alignment check
18	machine check
19-31	Intel reserved
32-255	maskable interrupts (devices & exceptions)

O mecanismo de interrupção torna eficiente a interação do processador com os dispositivos periféricos. Se não existissem interrupções, o processador perderia muito tempo "varrendo" todos os dispositivos do sistema para verificar se há eventos a serem tratados. Além disso, as interrupções permitem construir funções de entrada/saída assíncronas, ou seja, o processador não precisa esperar a conclusão de cada operação solicitada a um dispositivo, pois o dispositivo gera uma interrupção para "avisar" o processador quando a operação for concluída. Interrupções não são raras, pelo contrário: em um computador pessoal, o processador trata de centenas a milhares de interrupções por segundo, dependendo da carga do sistema e dos periféricos instalados.

5.2 Proteção do núcleo

Um sistema operacional deve gerenciar os recursos do hardware, fornecendo-os às aplicações conforme suas necessidades. Para assegurar a integridade dessa gerência, é essencial garantir que as aplicações não consigam acessar o hardware diretamente, mas sempre através de pedidos ao sistema operacional, que avalia e intermedeia todos os acessos ao hardware. Mas como impedir as aplicações de acessar o hardware diretamente?

Núcleo, drivers, utilitários e aplicações são constituídos basicamente de código de máquina. Todavia, devem ser diferenciados em sua capacidade de interagir com o hardware: enquanto o núcleo e os drivers devem ter pleno acesso ao hardware, para poder configurá-lo e gerenciá-lo, os utilitários e os aplicativos devem ter acesso mais restrito a ele, para não interferir nas configurações e na gerência, o que acabaria desestabilizando o sistema inteiro. Além disso, aplicações com acesso pleno ao hardware tornariam inúteis os mecanismos de segurança e controle de acesso aos recursos (tais como arquivos, diretórios e áreas de memória).

Para permitir diferenciar os privilégios de execução dos diferentes tipos de software, os processadores modernos contam com dois ou mais *níveis de privilégio de execução*. Esses níveis são controlados por flags especiais nos processadores, e as formas de mudança de um nível de execução para outro são controladas estritamente pelo processador. O processador Pentium, por exemplo, conta com 4 níveis de privilégio (sendo 0 o nível mais privilegiado), embora a maioria dos sistemas operacionais construídos para esse processador só use os níveis extremos (0 para o núcleo e drivers do sistema operacional e 3 para utilitários e aplicações). Na forma mais simples desse esquema, podemos considerar dois níveis básicos de privilégio:

Nível núcleo : também denominado nível *supervisor*, *sistema*, *monitor* ou ainda *kernel space*. Para um código executando nesse nível, todo o processador está acessível: todos os recursos internos do processador (registradores e portas de entrada/saída) e áreas de memória podem ser acessados. Além disso, todas as instruções do processador podem ser executadas. Ao ser ligado, o processador entra em operação neste nível.

Nível usuário (ou *userspace*): neste nível, somente um sub-conjunto das instruções do processador, registradores e portas de entrada/saída estão disponíveis. Instruções "perigosas" como `HALT` (parar o processador) e `RESET` (reiniciar o processador) são proibidas para todo código executando neste nível. Além disso, o hardware restringe o uso da memória, permitindo o acesso somente a áreas previamente definidas. Caso o código em execução tente executar uma instrução proibida ou acessar uma área de memória inacessível, o hardware irá gerar uma exceção, desviando a execução para uma rotina de tratamento dentro do núcleo, que provavelmente irá abortar o programa em execução (e também gerar a famosa frase "este programa executou uma instrução ilegal e será finalizado", no caso do Windows).

É fácil perceber que, em um sistema operacional convencional, o núcleo e os drivers operam no nível núcleo, enquanto os utilitários e as aplicações operam no nível usuário, confinados em áreas de memória distintas, conforme ilustrado na figura 6. Todavia, essa separação nem sempre segue uma regra tão simples; outras opções de organização de sistemas operacionais serão abordadas na seção 6.

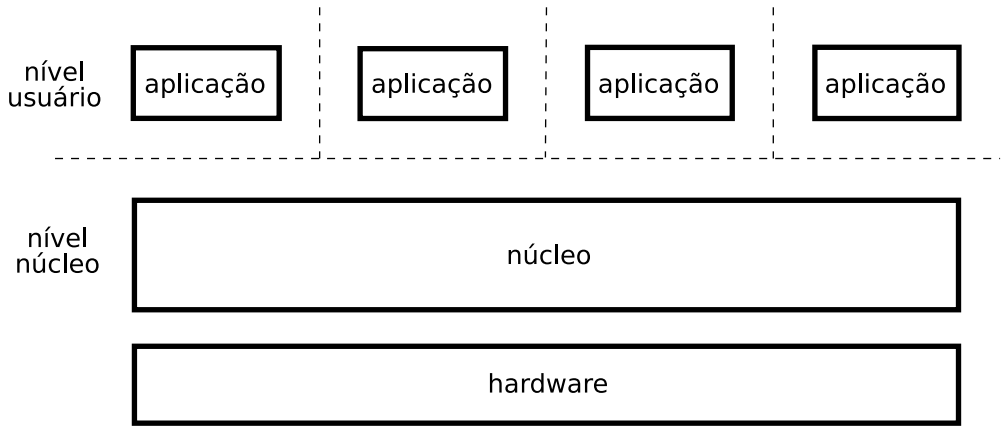

Figura 6: Separação entre o núcleo e as aplicações

5.3 Chamadas de sistema

O confinamento de cada aplicação em sua área de memória, imposto pelos mapeamentos de memória realizados pela MMU nos acessos em nível usuário, provê robustez e confiabilidade ao sistema, pois garante que uma aplicação não poderá interferir nas áreas de memória de outras aplicações ou do núcleo. Entretanto, essa proteção introduz um novo problema: como chamar, a partir de uma aplicação, as rotinas oferecidas pelo núcleo para o acesso ao hardware e suas abstrações? Em outras palavras, como uma aplicação pode acessar a placa de rede para enviar/receber dados, se não tem privilégio para acessar as portas de entrada/saída correspondentes nem pode invocar o código do núcleo que implementa esse acesso (pois esse código reside em outra área de memória)?

A resposta a esse problema está no mecanismo de interrupção, apresentado na seção 5.1. Os processadores implementam uma instrução especial que permite acionar o mecanismo de interrupção de forma intencional, sem depender de eventos externos ou internos. Ao ser executada, essa instrução (`int` no Pentium, `syscall` no MIPS) comuta o processador para o nível privilegiado e procede de forma similar ao tratamento de uma interrupção. Por essa razão, esse mecanismo é denominado *interrupção de software*, ou *trap*. Processadores modernos oferecem instruções específicas para entrar/sair do modo privilegiado, como `SYSCALL` e `SYSRET` (nos processadores Pentium), que permitem a transferência rápida do controle para o núcleo, com custo menor que o tratamento de uma interrupção.

A ativação de procedimentos do núcleo usando interrupções de software (ou outros mecanismos correlatos) é denominada *chamada de sistema* (*system call* ou *syscall*). Os sistemas operacionais definem chamadas de sistema para todas as operações envolvendo o acesso a recursos de baixo nível (periféricos, arquivos, alocação de memória, etc) ou abstrações lógicas (criação e finalização de tarefas, operadores de sincronização e comunicação, etc). Geralmente as chamadas de sistema são oferecidas para as aplicações em modo usuário através de uma *biblioteca do sistema* (*system library*), que prepara os parâmetros, invoca a interrupção de software e retorna à aplicação os resultados obtidos.

A figura 7 ilustra o funcionamento básico de uma chamada de sistema (a chamada `read`, que lê dados de um arquivo previamente aberto). Os seguintes passos são realizados:

1. No nível usuário, a aplicação invoca a função `read(fd, &buffer, bytes)` da biblioteca de sistema (no Linux é a biblioteca *GNU C Library*, ou `glibc`; no Windows, essas funções são implementadas pela API Win32).

2. A função `read` preenche uma área de memória com os parâmetros recebidos e escreve o endereço dessa área em um registrador da CPU. Em outro registrador, ela escreve o código da chamada de sistema desejada (no caso do Linux, seria `03h` para a *syscall* `read`).

3. A função `read` invoca uma interrupção de software (no caso do Linux, sempre é invocada a interrupção `80h`).

4. O processador comuta para o nível privilegiado (*kernel level*) e transfere o controle para a rotina apontada pela entrada `80h` do vetor de interrupções.

5. A rotina obtém o endereço dos parâmetros, verifica a validade de cada um deles e realiza (ou agenda para execução posterior) a operação desejada pela aplicação.

6. Ao final da execução da rotina, eventuais valores de retorno são escritos na área de memória da aplicação e o processamento retorna à função `read`, em modo usuário.

7. A função `read` finaliza sua execução e retorna o controle à aplicação.

8. Caso a operação solicitada não possa ser realizada imediatamente, a rotina de tratamento da interrupção de software passa o controle para a gerência de atividades, ao invés de retornar diretamente da interrupção de software para a aplicação solicitante. Isto ocorre, por exemplo, quando é solicitada a leitura de uma entrada do teclado.

9. Na seqüência, a gerência de atividades devolve o controle do processador a outra aplicação que também esteja aguardando o retorno de uma interrupção de software, e cuja operação solicitada já tenha sido concluída.

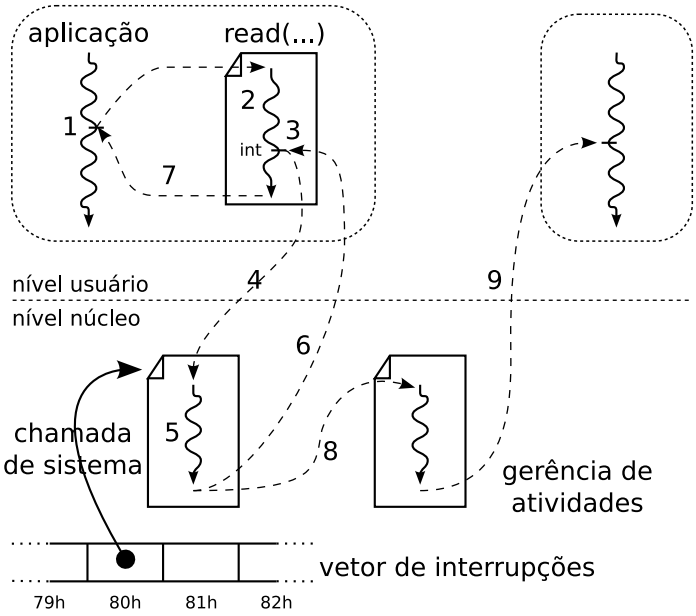

Figura 7: Roteiro típico de uma chamada de sistema

A maioria dos sistemas operacionais implementa centenas de chamadas de sistema distintas, para as mais diversas finalidades. O conjunto de chamadas de sistema oferecidas por um núcleo define a API (*Application Programming Interface*) desse sistema operacional. Exemplos de APIs bem conhecidas são a *Win32*, oferecida pelos sistemas Microsoft derivados do Windows NT, e a API *POSIX* [Gallmeister, 1994], que define um padrão de interface de núcleo para sistemas UNIX.

6 Arquiteturas de Sistemas Operacionais

Embora a definição de níveis de privilégio (seção 5.3) imponha uma estruturação mínima a um sistema operacional, as múltiplas partes que compõem o sistema podem ser organizadas de diversas formas, separando suas funcionalidades e modularizando seu projeto. Nesta seção serão apresentadas as arquiteturas mais populares para a organização de sistemas operacionais.

6.1 Sistemas monolíticos

Em um sistema monolítico, todos os componentes do núcleo operam em modo núcleo e se inter-relacionam conforme suas necessidades, sem restrições de acesso entre si, pois o código no nível núcleo tem acesso pleno a todos os recursos e áreas de memória. A figura 8 ilustra essa arquitetura.

A grande vantagem dessa arquitetura é seu desempenho: qualquer componente do núcleo pode acessar os demais componentes, toda a memória ou mesmo dispositivos

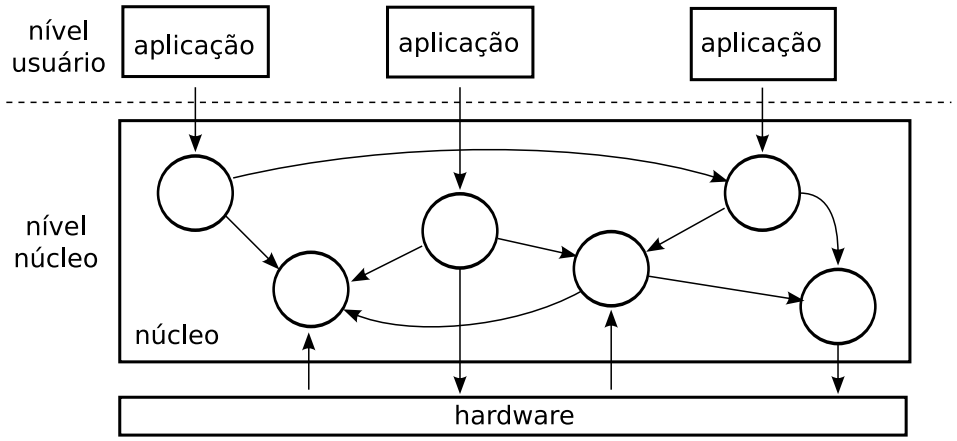

Figura 8: Uma arquitetura monolítica

periféricos diretamente, pois não há barreiras impedindo esse acesso. A interação direta entre componentes também leva a sistemas mais compactos.

Todavia, a arquitetura monolítica pode pagar um preço elevado por seu desempenho: a robustez e a facilidade de desenvolvimento. Caso um componente do núcleo perca o controle devido a algum erro, esse problema pode se alastrar rapidamente por todo o núcleo, levando o sistema ao colapso (travamento, reinicialização ou funcionamento errático). Além disso, a manutenção e evolução do núcleo se tornam mais complexas, porque as dependências e pontos de interação entre os componentes podem não ser evidentes: pequenas alterações na estrutura de dados de um componente podem ter um impacto inesperado em outros componentes, caso estes acessem aquela estrutura diretamente.

A arquitetura monolítica foi a primeira forma de organizar os sistemas operacionais; sistemas UNIX antigos e o MS-DOS seguiam esse modelo. Atualmente, apenas sistemas operacionais embutidos usam essa arquitetura, devido às limitações do hardware sobre o qual executam. O núcleo do Linux nasceu monolítico, mas vem sendo paulatinamente estruturado e modularizado desde a versão 2.0 (embora boa parte de seu código ainda permaneça no nível de núcleo).

6.2 Sistemas em camadas

Uma forma mais elegante de estruturar um sistema operacional faz uso da noção de camadas: a camada mais baixa realiza a interface com o hardware, enquanto as camadas intermediárias provêem níveis de abstração e gerência cada vez mais sofisticados. Por fim, a camada superior define a interface do núcleo para as aplicações (as chamadas de sistema). Essa abordagem de estruturação de software fez muito sucesso no domínio das redes de computadores, através do modelo de referência OSI (*Open Systems Interconnection*) [Day, 1983], e também seria de se esperar sua adoção no domínio dos sistemas operacionais. No entanto, alguns inconvenientes limitam sua aceitação nesse contexto:

- O empilhamento de várias camadas de software faz com que cada pedido de uma aplicação demore mais tempo para chegar até o dispositivo periférico ou recurso a ser acessado, prejudicando o desempenho do sistema.
- Não é óbvio como dividir as funcionalidades de um núcleo de sistema operacional em camadas horizontais de abstração crescente, pois essas funcionalidades são inter-dependentes, embora tratem muitas vezes de recursos distintos.

Em decorrência desses inconvenientes, a estruturação em camadas é apenas parcialmente adotada hoje em dia. Muitos sistemas implementam uma camada inferior de abstração do hardware para interagir com os dispositivos (a camada *HAL – Hardware Abstraction Layer*, implementada no Windows NT e seus sucessores), e também organizam em camadas alguns sub-sistemas como a gerência de arquivos e o suporte de rede (seguindo o modelo OSI). Como exemplos de sistemas fortemente estruturados em camadas podem ser citados o IBM OS/2 e o MULTICS [Corbató and Vyssotsky, 1965].

6.3 Sistemas micro-núcleo

Uma outra possibilidade de estruturação consiste em retirar do núcleo todo o código de alto nível (normalmente associado às políticas de gerência de recursos), deixando no núcleo somente o código de baixo nível necessário para interagir com o hardware e criar as abstrações fundamentais (como a noção de atividade). Por exemplo, usando essa abordagem o código de acesso aos blocos de um disco rígido seria mantido no núcleo, enquanto as abstrações de arquivo e diretório seriam criadas e mantidas por um código fora do núcleo, executando da mesma forma que uma aplicação do usuário.

Por fazer os núcleos de sistema ficarem menores, essa abordagem foi denominada *micro-núcleo* (ou *μ-kernel*). Um micro-núcleo normalmente implementa somente a noção de atividade, de espaços de memória protegidos e de comunicação entre atividades. Todos os aspectos de alto nível, como políticas de uso do processador e da memória, o sistema de arquivos e o controle de acesso aos recursos são implementados fora do núcleo, em processos que se comunicam usando as primitivas do núcleo. A figura 9 ilustra essa abordagem.

Em um sistema micro-núcleo, as interações entre componentes e aplicações são feitas através de trocas de mensagens. Assim, se uma aplicação deseja abrir um arquivo no disco rígido, envia uma mensagem para o gerente de arquivos que, por sua vez, se comunica com o gerente de dispositivos para obter os blocos de dados relativos ao arquivo desejado. Os processos não podem se comunicar diretamente, devido às restrições impostas pelos mecanismos de proteção do hardware. Por isso, todas as mensagens são transmitidas através de serviços do micro-núcleo, como mostra a figura 9. Como os processos têm de solicitar "serviços" uns dos outros, para poder realizar suas tarefas, essa abordagem também foi denominada *cliente-servidor*.

O micro-núcleos foram muito investigados durante os anos 80. Dois exemplos clássicos dessa abordagem são os sistemas Mach [Rashid et al., 1989] e Chorus

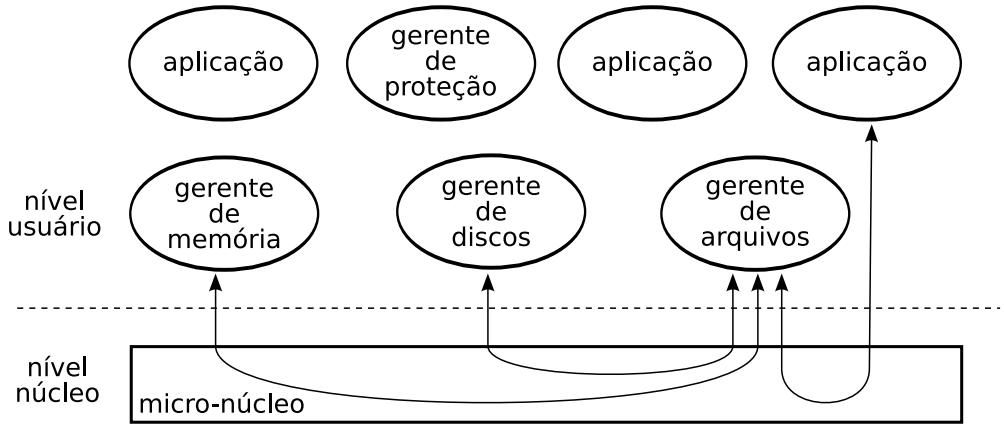

Figura 9: Visão geral de uma arquitetura micro-núcleo

[Rozier et al., 1992]. As principais vantagens dos sistemas micro-núcleo são sua robustez e flexibilidade: caso um sub-sistema tenha problemas, os mecanismos de proteção de memória e níveis de privilégio irão confiná-lo, impedindo que a instabilidade se alastre ao restante do sistema. Além disso, é possível customizar o sistema operacional, iniciando somente os componentes necessários ou escolhendo os componentes mais adequados às aplicações que serão executadas.

Vários sistemas operacionais atuais adotam parcialmente essa estruturação; por exemplo, o MacOS X da Apple tem suas raízes no sistema Mach, ocorrendo o mesmo com o Digital UNIX. Todavia, o custo associado às trocas de mensagens entre componentes pode ser bastante elevado, o que prejudica seu desempenho e diminui a aceitação desta abordagem. O QNX é um dos poucos exemplos de micro-núcleo amplamente utilizado, sobretudo em sistemas embutidos e de tempo-real.

6.4 Máquinas virtuais

Para que programas e bibliotecas possam executar sobre uma determinada plataforma computacional, é necessário que tenham sido compilados para ela, respeitando o conjunto de instruções do processador e o conjunto de chamadas do sistema operacional. Da mesma forma, um sistema operacional só poderá executar sobre uma plataforma de hardware se for compatível com ela. Nos sistemas atuais, as interfaces de baixo nível são pouco flexíveis: geralmente não é possível criar novas instruções de processador ou novas chamadas de sistema, ou mudar sua semântica. Por isso, um sistema operacional só funciona sobre o hardware para o qual foi construído, uma biblioteca só funciona sobre o hardware e sistema operacional para os quais foi projetada e as aplicações também têm de obedecer a interfaces pré-definidas.

Todavia, é possível contornar os problemas de compatibilidade entre os componentes de um sistema através de técnicas de *virtualização*. Usando os serviços oferecidos por um determinado componente do sistema, é possível construir uma camada de software que

ofereça aos demais componentes serviços com outra interface. Essa camada permitirá assim o acoplamento entre interfaces distintas, de forma que um programa desenvolvido para uma plataforma *A* possa executar sobre uma plataforma distinta *B*. O sistema computacional visto através dessa camada é denominado *máquina virtual*.

A figura 10, extraída de [Smith and Nair, 2004], mostra um exemplo de máquina virtual, onde uma camada de virtualização permite executar um sistema operacional Windows e suas aplicações sobre uma plataforma de hardware Sparc, distinta daquela para a qual foi projetado (Intel/AMD).

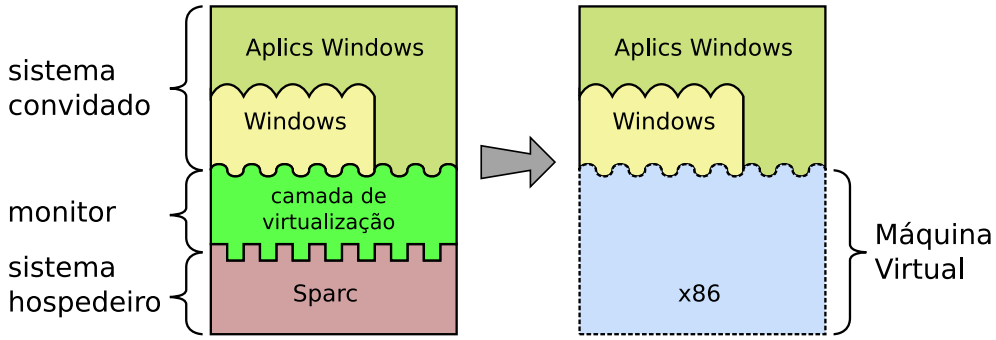

Figura 10: Uma máquina virtual [Smith and Nair, 2004].

Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na figura 10:

- O sistema real, ou sistema hospedeiro (*host system*), que contém os recursos reais de hardware e software do sistema;
- o sistema virtual, também denominado sistema convidado (*guest system*), que executa sobre o sistema virtualizado; em alguns casos, vários sistemas virtuais podem coexistir, executando sobre o mesmo sistema real;
- a camada de virtualização, denominada *hipervisor* ou *monitor de virtualização* (*VMM - Virtual Machine Monitor*), que constrói as interfaces virtuais a partir da interface real.

Na década de 1970, os pesquisadores Popek & Goldberg formalizaram vários conceitos associados às máquinas virtuais, e definiram as condições necessárias para que uma plataforma de hardware suporte de forma eficiente a virtualização [Popek and Goldberg, 1974]. Nessa mesma época surgem as primeiras experiências concretas de utilização de máquinas virtuais para a execução de aplicações, com o ambiente *UCSD p-System*, no qual programas Pascal são compilados para execução sobre um hardware abstrato denominado *P-Machine*. Com o aumento de desempenho e funcionalidades do hardware PC e o surgimento da linguagem Java, no início dos anos 90, o interesse pelas tecnologias de virtualização voltou à tona. Atualmente, as soluções de virtualização de linguagens e de plataformas vêm despertando grande interesse

do mercado. Várias linguagens são compiladas para máquinas virtuais portáveis e os processadores mais recentes trazem um suporte nativo à virtualização de hardware, finalmente respeitando as condições conceituais definidas no início dos anos 70.

Existem diversas possibilidades de implementação de sistemas de máquinas virtuais. De acordo com o tipo de sistema convidado suportado, os ambientes de máquinas virtuais podem ser divididos em duas grandes famílias (figura 11):

Máquinas virtuais de aplicação : são ambientes de máquinas virtuais destinados a suportar apenas um processo ou aplicação convidada específica. A máquina virtual Java é um exemplo desse tipo de ambiente.

Máquinas virtuais de sistema : são construídos para suportar sistemas operacionais convidados completos, com aplicações convidadas executando sobre eles. Como exemplos podem ser citados os ambientes *VMWare*, *Xen* e *VirtualBox*.

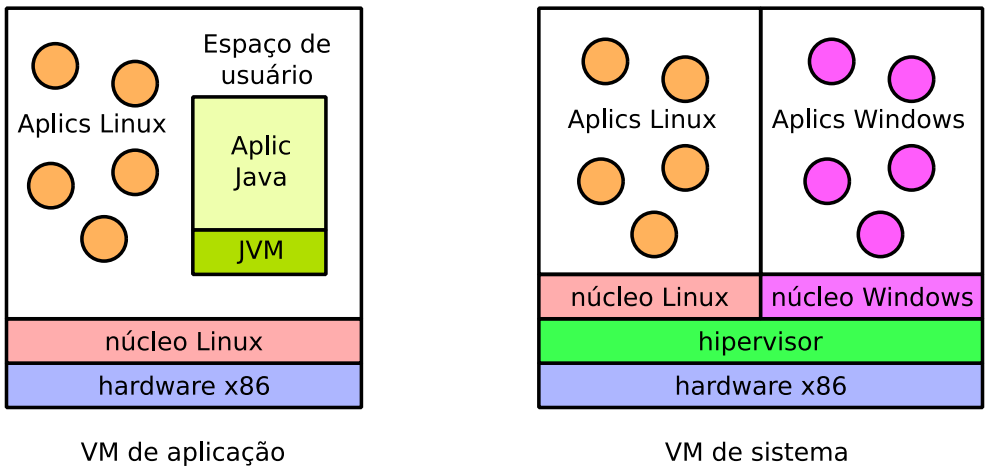

Figura 11: Máquinas virtuais de aplicação e de sistema.

As máquinas virtuais de aplicação são geralmente usadas como suporte de execução de linguagens de programação. As primeiras experiências com linguagens usando máquinas virtuais remontam aos anos 1970, com a linguagem *UCSD Pascal* (da Universidade da Califórnia em San Diego). Na época, um programa escrito em Pascal era compilado em um código binário denominado *P-Code*, que executava sobre o processador abstrato *P-Machine*. O interpretador de *P-Codes* era bastante compacto e facilmente portável, o que tornou o sistema P muito popular. Hoje, muitas linguagens adotam estratégias similares, como Java, C#, Python, Perl, Lua e Ruby. Em C#, o código-fonte é compilado em um formato intermediário denominado CIL (*Common Intermediate Language*), que executa sobre uma máquina virtual CLR (*Common Language Runtime*). CIL e CLR fazem parte da infraestrutura .NET da Microsoft.

Máquinas virtuais de sistema suportam um ou mais sistemas operacionais convidados, com suas respectivas aplicações, que executam de forma isolada e independente.

Em uma máquina virtual, cada sistema operacional convidado tem a ilusão de executar sozinho sobre uma plataforma de hardware exclusiva. Como o sistema operacional convidado e o ambiente de execução dentro da máquina virtual são idênticos ao da máquina real, é possível usar os softwares já construídos para a máquina real dentro das máquinas virtuais. Essa transparência evita ter de construir novas aplicações ou adaptar as já existentes.

As máquinas virtuais de sistema constituem a primeira abordagem usada para a construção de hipervisores, desenvolvida na década de 1960. No que diz respeito à sua arquitetura, existem basicamente dois tipos de hipervisores de sistema, apresentados na figura 12:

Hipervisores nativos (ou de tipo I): o hipervisor executa diretamente sobre o hardware da máquina real, sem um sistema operacional subjacente. A função do hipervisor é multiplexar os recursos de hardware (memória, discos, interfaces de rede, etc) de forma que cada máquina virtual veja um conjunto de recursos próprio e independente. Alguns exemplos de sistemas que empregam esta abordagem são o *IBM OS/370*, o *VMWare ESX Server* e o ambiente *Xen*.

Hipervisores convidados (ou de tipo II): o hipervisor executa como um processo normal sobre um sistema operacional hospedeiro. O hipervisor usa os recursos oferecidos pelo sistema operacional real para oferecer recursos virtuais ao sistema operacional convidado que executa sobre ele. Exemplos de sistemas que adotam esta estrutura incluem o *VMWare Workstation*, o *QEmu* e o *VirtualBox*.

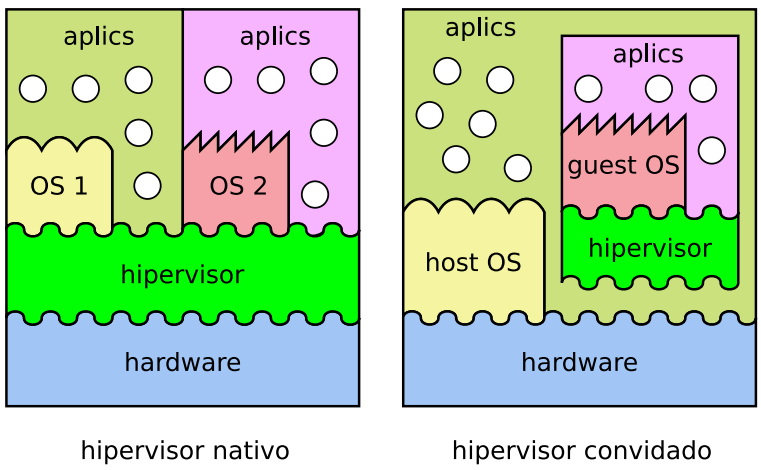

Figura 12: Arquiteturas de máquinas virtuais de sistema.

Os trabalhos [Goldberg, 1973, Blunden, 2002] relacionam algumas vantagens para a utilização de máquinas virtuais em sistemas de computação:

- Aperfeiçoamento e testes de novos sistemas operacionais;

- Ensino prático de sistemas operacionais e programação de baixo nível;
- Executar diferentes sistemas operacionais sobre o mesmo hardware, simultaneamente;
- Simular configurações e situações diferentes do mundo real, como por exemplo, mais memória disponível, outros dispositivos de E/S;
- Simular alterações e falhas no hardware para testes ou reconfiguração de um sistema operacional, provendo confiabilidade e escalabilidade para as aplicações;
- Garantir a portabilidade das aplicações legadas (que executariam sobre uma VM simulando o sistema operacional original);
- Desenvolvimento de novas aplicações para diversas plataformas, garantindo a portabilidade destas aplicações;
- Diminuir custos com hardware.

A principal desvantagem do uso de máquinas virtuais é o custo adicional de execução dos processos na máquina virtual em comparação com a máquina real. Esse custo é muito variável, podendo passar de 50% em plataformas sem suporte de hardware à virtualização, como os PCs de plataforma Intel mais antigos [Dike, 2000, Blunden, 2002]. Todavia, pesquisas recentes têm obtido a redução desse custo a patamares abaixo de 20%, graças sobretudo a ajustes no código do sistema hospedeiro [King et al., 2003]. Esse problema não existe em ambientes cujo hardware oferece suporte à virtualização, como é o caso dos mainframes e dos processadores Intel/AMD mais recentes.

7 Um breve histórico dos sistemas operacionais

Os primeiros sistemas de computação, no final dos anos 40 e início dos anos 50, não possuíam sistema operacional. Por outro lado, os sistemas de computação atuais possuem sistemas operacionais grandes, complexos e em constante evolução. A seguir são apresentados alguns dos marcos mais relevantes na história dos sistemas operacionais [Foundation, 2005]:

Anos 40 : cada programa executava sozinho e tinha total controle do computador. A carga do programa em memória, a varredura dos periféricos de entrada para busca de dados, a computação propriamente dita e o envio dos resultados para os periférico de saída, byte a byte, tudo devia ser programado detalhadamente pelo desenvolvedor da aplicação.

Anos 50 : os sistemas de computação fornecem "bibliotecas de sistema" (*system libraries*) que encapsulam o acesso aos periféricos, para facilitar a programação de aplicações. Algumas vezes um programa "monitor" (*system monitor*) auxilia a carga

e descarga de aplicações e/ou dados entre a memória e periféricos (geralmente leitoras de cartão perfurado, fitas magnéticas e impressoras de caracteres).

1961 : o grupo do pesquisador Fernando Corbató, do MIT, anuncia o desenvolvimento do CTSS – *Compatible Time-Sharing System [Corbató et al., 1962]*, o primeiro sistema operacional com compartilhamento de tempo.

1965 : a IBM lança o OS/360, um sistema operacional avançado, com compartilhamento de tempo e excelente suporte a discos.

1965 : um projeto conjunto entre MIT, GE e Bell Labs define o sistema operacional *Multics*, cujas idéias inovadoras irão influenciar novos sistemas durante décadas.

1969 : Ken Thompson e Dennis Ritchie, pesquisadores dos Bell Labs, criam a primeira versão do UNIX.

1981 : a Microsoft lança o MS-DOS, um sistema operacional comprado da empresa *Seattle Computer Products* em 1980.

1984 : a Apple lança o sistema operacional Macintosh OS 1.0, o primeiro a ter uma interface gráfica totalmente incorporada ao sistema.

1985 : primeira tentativa da Microsoft no campo dos sistemas operacionais com interface gráfica, através do MS-Windows 1.0.

1987 : Andrew Tanenbaum, um professor de computação holandês, desenvolve um sistema operacional didático simplificado, mas respeitando a API do UNIX, que foi batizado como *Minix*.

1987 : IBM e Microsoft apresentam a primeira versão do OS/2, um sistema multitarefa destinado a substituir o MS-DOS e o Windows. Mais tarde, as duas empresas rompem a parceria; a IBM continua no OS/2 e a Microsoft investe no ambiente Windows.

1991 : Linus Torvalds, um estudante de graduação finlandês, inicia o desenvolvimento do Linux, lançando na rede Usenet o núcleo 0.01, logo abraçado por centenas de programadores ao redor do mundo.

1993 : a Microsoft lança o Windows NT, o primeiro sistema 32 bits da empresa.

1993 : lançamento dos UNIX de código aberto FreeBSD e NetBSD.

2001 : a Apple lança o MacOS X, um sistema operacional derivado da família UNIX BSD.

2001 : lançamento do Windows XP.

2004 : lançamento do núcleo Linux 2.6.

2006 : lançamento do Windows Vista.

Esse histórico reflete apenas o surgimento de alguns sistemas operacionais relativamente populares; diversos sistemas acadêmicos ou industriais de grande importância pelas contribuições inovadoras, como *Mach*, *Chorus*, *QNX* e *Plan 9*, não estão representados.

Questões

1. Quais os dois principais objetivos dos sistemas operacionais?
2. Por que a abstração de recursos é importante para os desenvolvedores de aplicações? Ela tem utilidade para os desenvolvedores do próprio sistema operacional?
3. A gerência de atividades permite compartilhar o processador, executando mais de uma aplicação ao mesmo tempo. Identifique as principais vantagens trazidas por essa funcionalidade e os desafios a resolver para implementá-la.
4. O que caracteriza um sistema operacional de tempo real? Quais as duas classificações de sistemas operacionais de tempo real e suas diferenças?
5. O que diferencia o *núcleo* do restante do sistema operacional?
6. Seria possível construir um sistema operacional seguro usando um processador que não tenha níveis de privilégio? Por que?
7. O processador Pentium possui dois bits para definir o nível de privilégio, resultando em 4 níveis distintos. A maioria dos sistemas operacionais para esse processador usa somente os níveis extremos (0 e 3, ou 00_2 e 11_2). Haveria alguma utilidade para os níveis intermediários?
8. Quais as diferenças entre *interrupções*, *exceções* e *traps*?
9. Quais as implicações de mascarar interrupções? O que pode ocorrer se o processador ignorar interrupções por muito tempo? O que poderia ser feito para evitar o mascaramento de interrupções?
10. O comando em linguagem C `fopen` é uma chamada de sistema ou uma função de biblioteca? Por que?
11. Monte uma tabela com os benefícios e deficiências mais significativos das principais arquiteturas de sistemas operacionais.
12. O Linux possui um núcleo similar com o da figura 8, mas também possui "tarefas de núcleo" que executam como os gerentes da figura 9. Seu núcleo é monolítico ou micro-núcleo? Por que?

Exercícios

1. Relacione as afirmações aos respectivos tipos de sistemas operacionais: distribuído (D), multi-usuário (M), desktop (K), servidor (S), embutido (E) ou de tempo-real (T):

   [ ] Deve ter um comportamento temporal previsível, ou seja, com prazos de resposta bem definidos.

   [ ] A localização dos recursos do sistema é transparente para os usuários.

   [ ] Todos os recursos do sistema têm proprietários e existem regras controlando o acesso aos mesmos pelos usuários.

   [ ] A gerência de energia é muito importante neste tipo de sistema.

   [ ] Prioriza a gerência da interface gráfica, recursos multimídia e a interação com o usuário.

   [ ] Construído para gerenciar de forma eficiente grandes volumes de recursos.

   [ ] São sistemas operacionais compactos, construídos para executar sobre plataformas com poucos recursos.

2. A operação em modo usuário permite ao processador executar somente parte das instruções disponíveis em seu conjunto de instruções. Quais das seguintes operações não deveria ser permitida em nível usuário? Por que?

   (a) Ler uma porta de entrada/saída

   (b) Efetuar uma divisão inteira

   (c) Escrever um valor em uma posição de memória

   (d) Ajustar o valor do relógio do hardware

   (e) Ler o valor dos registradores do processador

   (f) Mascarar uma ou mais interrupções

3. Indique quais das seguintes operações teriam de ser implementadas por chamadas de sistema, justificando suas respostas:

   (a) Ler o relógio do hardware

   (b) Enviar um pacote de rede

   (c) Calcular um logaritmo natural

   (d) Obter um número aleatório

   (e) Remover um arquivo

4. Coloque na ordem correta as ações abaixo, que ocorrem durante a execução da função `printf("Hello world\n")` por um processo (observe que algumas dessas ações podem ou não fazer parte da seqüência).

   [ ] A rotina de tratamento da interrupção de software, dentro do núcleo, é ativada.

   [ ] Os valores de retorno da chamada de sistema são devolvidos ao processo.

   [ ] A função da biblioteca processa os parâmetros de entrada `Hello world\n`.

   [ ] A função `printf` ajusta os registradores para solicitar a chamada de sistema `write()`

   [ ] O disco gera uma interrupção indicando a conclusão da operação.

   [ ] O escalonador escolhe o processo mais prioritário.

   [ ] Uma interrupção de software é executada.

   [ ] O processo chama a função `printf()` da biblioteca do sistema

   [ ] A operação de escrita no terminal é agendada pela rotina da interrupção.

   [ ] O controle volta para a função `printf`, em modo usuário.

Projetos

1. O utilitário `strace` do UNIX permite observar a seqüência de chamadas de sistema efetuadas por uma aplicação. Em um terminal UNIX, execute `strace date` para descobrir quais os arquivos abertos pela execução do utilitário `date` (que indica a data e hora correntes). Por que o utilitário `date` precisa fazer chamadas de sistema?

2. O utilitário `ltrace` do UNIX permite observar a seqüência de chamadas de biblioteca efetuadas por uma aplicação. Em um terminal UNIX, execute `ltrace date` para descobrir as funções de biblioteca chamadas pela execução do utilitário `date` (que indica a data e hora correntes). Pode ser observada alguma relação entre as chamadas de biblioteca e as chamadas de sistema observadas no ítem anterior?

Referências

[Arpaci-Dusseau et al., 2003] Arpaci-Dusseau, A., Arpaci-Dusseau, R., Burnett, N., Denehy, T., Engle, T., Gunawi, H., Nugent, J., and Popovici, F. (2003). Transforming policies into mechanisms with InfoKernel. In *19th ACM Symposium on Operating Systems Principles*.

[Blunden, 2002] Blunden, B. (2002). *Virtual Machine Design and Implementation in C/C++*. Worldware Publishing.

[Corbató et al., 1962] Corbató, F., Daggett, M., and Daley, R. (1962). An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference*.

[Corbató and Vyssotsky, 1965] Corbató, F. J. and Vyssotsky, V. A. (1965). Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196.

[Dasgupta et al., 1991] Dasgupta, P., Richard J. LeBlanc, J., Ahamad, M., and Ramachandran, U. (1991). The Clouds distributed operating system. *Computer*, 24(11):34–44.

[Day, 1983] Day, J. (1983). The OSI reference model. *Proceedings of the IEEE*.

[Dike, 2000] Dike, J. (2000). A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*.

[Foundation, 2005] Foundation, W. (2005). Wikipedia online enciclopedia. http://www.wikipedia.org.

[Gallmeister, 1994] Gallmeister, B. (1994). *POSIX.4: Programming for the Real World*. O'Reilly.

[Goldberg, 1973] Goldberg, R. (1973). Architecture of virtual machines. In *AFIPS National Computer Conference*.

[King et al., 2003] King, S., Dunlap, G., and Chen, P. (2003). Operating system support for virtual machines. In *Proceedings of the USENIX Technical Conference*.

[Patterson and Henessy, 2005] Patterson, D. and Henessy, J. (2005). *Organização e Projeto de Computadores*. Campus.

[Pike et al., 1993] Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. (1993). The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76.

[Popek and Goldberg, 1974] Popek, G. and Goldberg, R. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.

[Rashid et al., 1989] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., and Jones, M. B. (1989). Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA. IEEE Comput. Soc. Press.

[Rozier et al., 1992] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W. (1992). Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA).

[Silberschatz et al., 2001] Silberschatz, A., Galvin, P., and Gane, G. (2001). *Sistemas Operacionais – Conceitos e Aplicações*. Campus.

[Smith and Nair, 2004] Smith, J. and Nair, R. (2004). *Virtual Machines: Architectures, Implementations and Applications*. Morgan Kaufmann.

[Tanenbaum, 2003] Tanenbaum, A. (2003). *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall.

[Tanenbaum et al., 1991] Tanenbaum, A., Kaashoek, M., van Renesse, R., and Bal, H. (1991). The Amoeba distributed operating system – a status report. *Computer Communications*, 14:324–335.