Capítulo 2

Estrutura de um SO

Este capítulo apresenta os principais componentes de uma sistema operacional e os mecanismos de hardware necessários para sua implementação.

2.1 Elementos do 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 aplicativos e programas utilitários.

Código de inicialização: (*boot code*) 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.

Drivers: módulos de código específicos para acessar os dispositivos físicos. Existe um driver para cada tipo de dispositivo, como discos rígidos SATA, portas USB, placas gráfica, 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.

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 estão relacionadas entre si conforme apresentado na Figura 2.1. Na figura, a região cinza indica o sistema operacional propriamente dito. A forma como os diversos componentes do SO são interligados e se relacionam varia de sistema para sistema; algumas possibilidades de organização são discutidas no Capítulo 3.

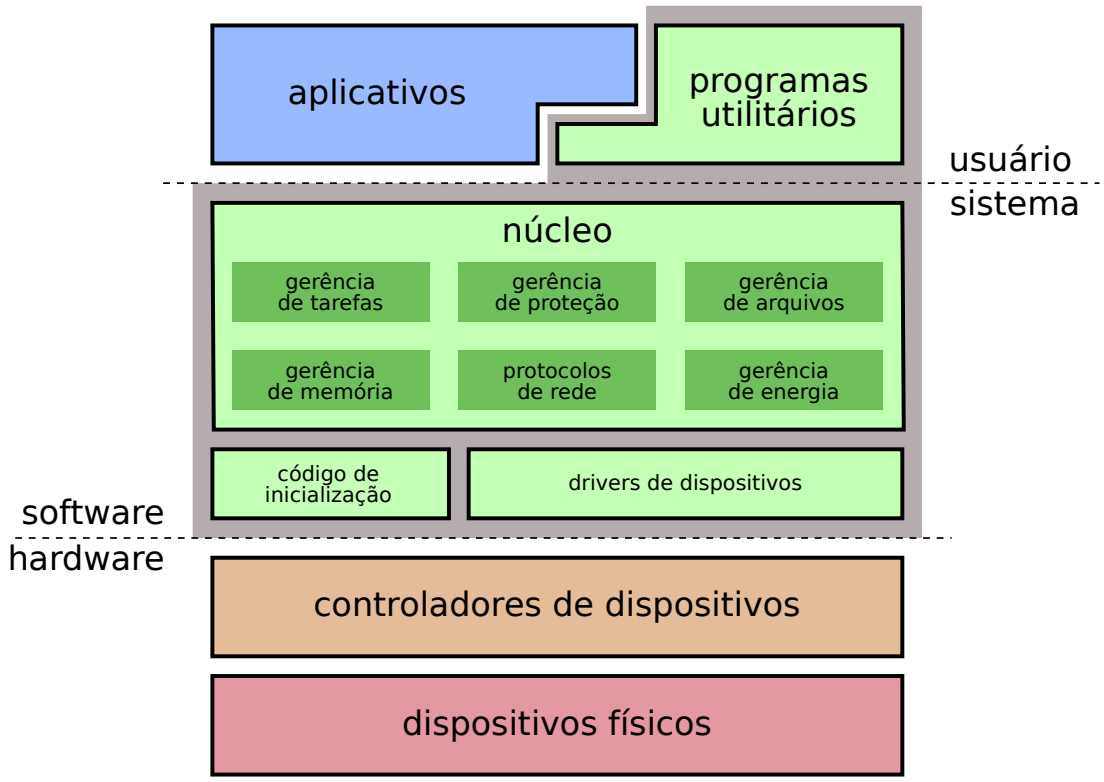

Figura 2.1: Estrutura de um sistema operacional típico

A camada mais baixa do sistema operacional, que constitui o chamado "núcleo" do sistema (ou (*kernel*), usualmente executa em um modo especial de operação do processador, denominado *modo privilegiado* ou modo sistema (vide Seção 2.2.3). Os demais programas e aplicações executam em um modo denominado *modo usuário*.

Sistemas operacionais reais têm estruturas que seguem aquela apresentada na Figura 2.1, embora sejam geralmente muito mais complexas. Por exemplo, o sistema operacional Android, usado em telefones celulares, é baseado no Linux, mas tem uma rica estrutura de bibliotecas e serviços no nível de usuário, que são usados para a construção de aplicações. A Figura 2.2 representa a arquitetura de um sistema Android 8.0.

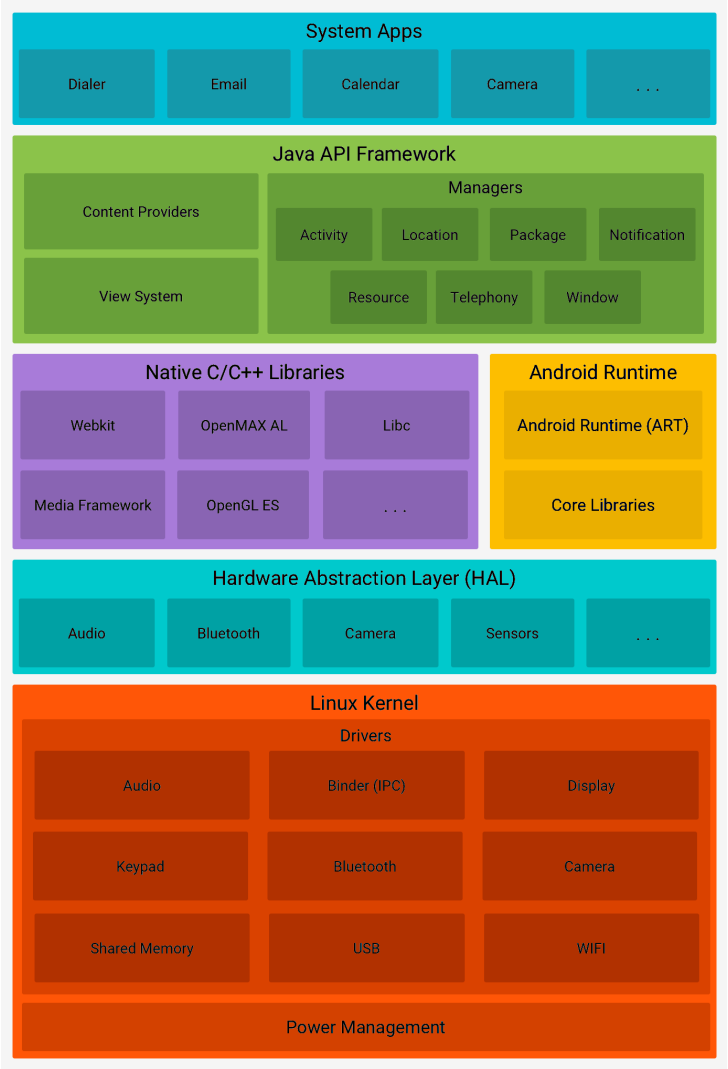

Figura 2.2: Estrutura de um sistema operacional Android [Google, 2018].

2.2 Elementos de hardware

O sistema operacional executa diretamente sobre o hardware, abstraindo e gerenciando recursos para as aplicações. Para que o SO possa cumprir suas funções com eficiência e confiabilidade, o hardware deve prover algumas funcionalidades básicas que são discutidas nesta seção.

2.2.1 Arquitetura do computador

Um computador típico é constituído de um ou mais processadores, responsáveis pela execução das instruções das aplicações, uma ou mais áreas de memórias que armazenam 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, terminais e teclados.

A maioria dos computadores monoprocessados 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 ideia 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 2.3 ilustra a arquitetura de um computador típico.

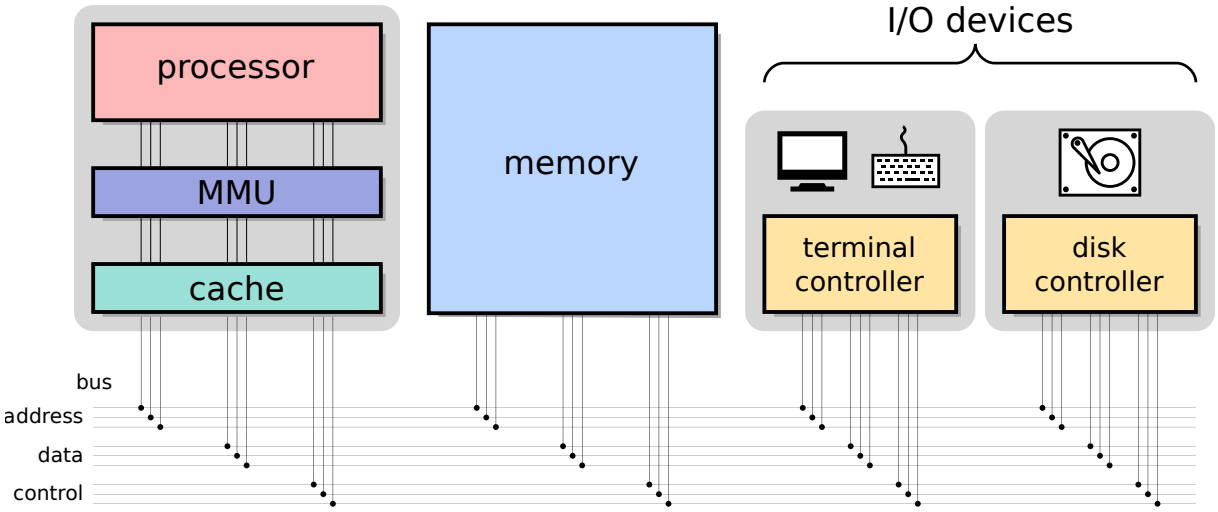

Figura 2.3: Arquitetura de um computador típico

O centro do sistema de computação é o processador. Ele é responsável por continuamente ler instruções e dados da memória ou dos periféricos, processá-los e enviar os resultados de volta à memória ou a outros periféricos. Em sua forma mais simples, um processador convencional é 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.).

Processadores modernos são incrivelmente mais complexos, podendo possuir diversos núcleos de processamento (os chamados *cores*), cada um contendo vários processadores lógicos internos (*hyperthreading*). Além disso, um computador pode conter vários processadores trabalhando em paralelo. Outro aspecto diz respeito à memória, que pode comportar vários níveis de cache, dentro do processador e na placa-mãe. Uma descrição mais detalhada da arquitetura de computadores modernos pode ser encontrada em [Stallings, 2010].

Todas as transferências 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 escrita) e o **barramento de dados** transporta a informação a ser transferida entre o processador e a memória ou um controlador de dispositivo.

O acesso do processador à memória é mediado por um controlador específico (que pode estar fisicamente dentro do próprio chip do processador): a **Unidade de Gerência de Memória** (MMU - *Memory Management Unit*). Ela é responsável por analisar cada endereço de memória acessado pelo processador, validá-lo, efetuar conversões de endereçamento porventura necessárias e executar a operação solicitada pelo processador (leitura ou escrita de uma posição de memória). Uma memória *cache* permite armazenar os dados mais recentemente lidos da memória, para melhorar o desempenho.

Os periféricos do computador (discos, teclado, monitor, etc.) são acessados através de circuitos eletrônicos específicos, 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. Dentro do computador, 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 2.1 a seguir apresenta alguns endereços portas de entrada/saída para acessar controladores em um PC típico (podem variar):

Dispositivo	**Endereços de acesso**
temporizador	0040-0043
teclado	0060-006F
porta serial COM1	02F8-02FF
controlador SATA	30BC-30BF
controlador Ethernet	3080-309F
controlador	3000-303F

Tabela 2.1: Endereços de acesso a dispositivos (em hexadecimal).

2.2.2 Interrupções e exceções

A comunicação entre o processador e os dispositivos se dá através do acesso às portas de entrada/saída, que podem ser lidas e/ou escritas pelo processsador. Esse acesso é feito por iniciativa do processador, quando este precisa ler ou escrever dados no dispositivo. Entretanto, muitas vezes um dispositivo precisa informar o processador rapidamente sobre um evento interno, como a chegada de um pacote de rede, um clique de mouse ou a conclusão de uma operação de disco. Neste caso, o controlador tem duas alternativas:

- aguardar até que o processador o consulte, o que poderá ser demorado caso o processador esteja ocupado com outras tarefas (o que geralmente ocorre);
- notificar o processador, enviando a ele uma *requisição de interrupção* (IRQ – *Interrupt ReQuest*) através do barramento de controle.

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

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;

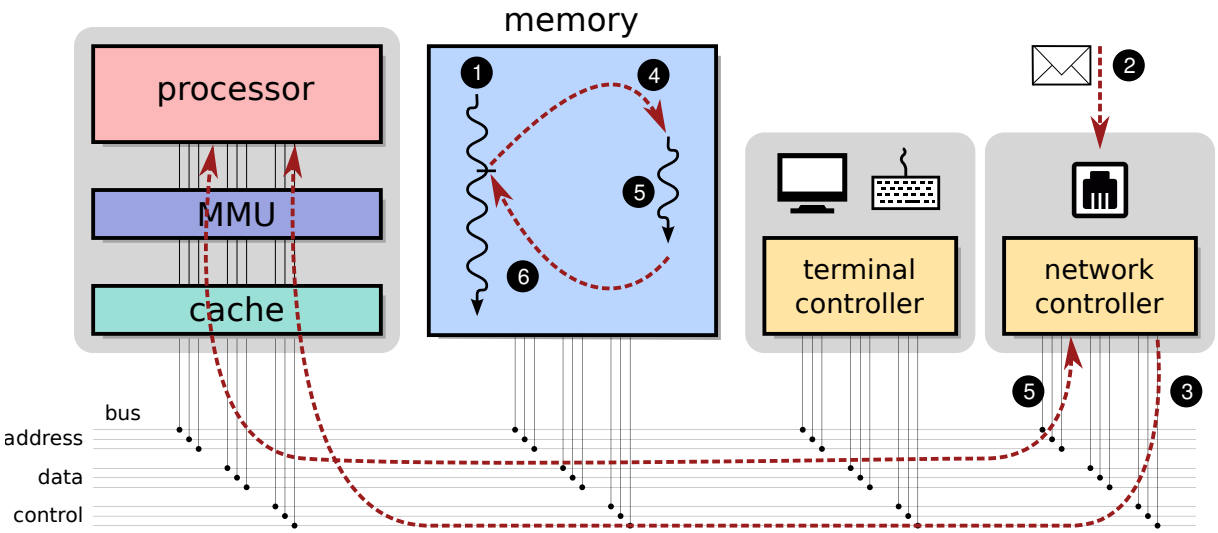

Figura 2.4: Roteiro típico de um tratamento de interrupção

3. o controlador Ethernet 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 interagir com o controlador de rede (via barramentos de dados e de endereços) para transferir os dados do pacote de rede do controlador para a memória;

6. a rotina de tratamento da interrupção é finalizada e o processador retorna à execução do programa que havia sido interrompido.

Essa sequência de ações ocorre a cada requisição de interrupção recebida pelo processador, que geralmente corresponde a um evento ocorrido em um dispositivo: a chegada de um pacote de rede, um click no mouse, uma operação concluída pelo disco, etc. Interrupções não são eventos raros, pelo contrário: centenas ou mesmo milhares de interrupções são recebidas pelo processador por segundo, dependendo da carga e da configuração do sistema (número e tipo dos periféricos). Assim, as rotinas de tratamento de interrupção devem ser curtas e realizar suas tarefas rapidamente, para não prejudicar o desempenho do sistema.

Para distinguir interrupções geradas por dispositivos distintos, cada interrupção é identificada pelo hardware por um número inteiro. 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 uma tabela de endereços de funções denominada *Tabela de Interrupções* (IVT - *Interrupt Vector Table*); cada entrada dessa tabela aponta para a rotina de tratamento da interrupção correspondente. Por exemplo, se a entrada 5 da tabela contém o valor `3C20h`, então a rotina de tratamento da IRQ 5 iniciará na posição `3C20h` da memória RAM. A tabela 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.

A maioria das 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*. Ações como instruções ilegais (inexistentes ou com operandos inválidos), tentativas 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 a mesma tabela de endereços de funções). A Tabela 2.2 representa parte da tabela de interrupções do processador Intel Pentium.

Tabela 2.2: Tabela 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. O Capítulo 19 traz informações mais detalhadas sobre o tratamento de interrupções pelo sistema operacional.

2.2.3 Níveis de privilégio

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 aplicativos e utilitários devem ter acesso mais restrito a ele, para não interferir nas configurações e na gerência, o que poderia desestabilizar o sistema inteiro. Além disso, aplicações com acesso pleno ao hardware seriam um risco à segurança, pois poderiam contornar facilmente os mecanismos de controle de acesso aos recursos (tais como arquivos e áreas de memória).

Para permitir diferenciar os privilégios de execução dos diferentes tipos de software, os processadores modernos implementam *níveis de privilégio de execução*. Esses níveis são controlados por flags especiais na CPU, que podem ser ajustados em função tipo de código em execução. Os processadores no padrão Intel x86, por exemplo, dispõem de 4 níveis de privilégio (0 . . . 3, sendo 0 o nível mais privilegiado), embora a maioria dos sistemas operacionais construídos para esses processadores só use os níveis extremos (0 para o núcleo do sistema operacional e 3 para utilitários e aplicações)[1].

Figura 2.5: Níveis de privilégio de processadores Intel x86

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, todas as funcionalidades do processador estão disponíveis: todas as instruções disponíveis podem ser executadas e todos os recursos internos (registradores e portas de entrada/saída) e áreas de memória podem ser acessados. Ao ser ligado durante a inicialização do computador, o processador entra em operação neste nível.

Nível usuário: neste nível, também chamado *userspace*, somente um subconjunto das instruções do processador e registradores estão disponíveis. Por exemplo, instruções consideradas "perigosas", como `RESET` (reiniciar o processador) e `IN/OUT` (acessar portas de entrada/saída), são proibidas. Caso o código em execução tente executar uma instrução proibida, o hardware irá gerar uma

[1] A ideia de definir níveis de privilégio para o código foi introduzida pelo sistema operacional Multics [Corbató and Vyssotsky, 1965], nos anos 1960. No Multics, os níveis eram organizados em camadas ao redor do núcleo, como em uma cebola, chamadas "anéis de proteção" (*protection rings*).

exceção, desviando a execução para uma rotina de tratamento dentro do núcleo. Essa rotina irá tratar o erro, provavelmente abortando o programa em execução[2].

Dessa forma, o núcleo do sistema operacional, bem como *drivers* e o código de inicialização, executam em modo núcleo, enquanto os programas utilitários e os aplicativos executam em modo usuário. Os flags que definem o nível de privilégio só podem ser modificados por código executando no nível núcleo, o que impede usuários maliciosos de contornar essa barreira de proteção.

Além da proteção oferecida pelos níveis de privilégio, o núcleo do sistema operacional pode configurar a unidade de gerência de memória (MMU) para criar uma área de memória exclusiva para cada aplicação, isolada das demais aplicações e do núcleo. As aplicações não podem contornar essa barreira de memória, pois a configuração da MMU só pode ser alterada em modo supervisor, ou seja, pelo próprio núcleo.

Com a proteção provida pelos níveis de privilégio do processador e pelas áreas de memória exclusivas configuradas pela MMU, obtém-se um modelo de execução confinada, no qual as aplicações executam isoladas umas das outras e do núcleo, conforme ilustrado na Figura 2.6. Esse modelo de execução é bastante robusto, pois isola os erros, falhas e comportamentos indevidos de uma aplicação do restante do sistema, proporcionando mais estabilidade e segurança.

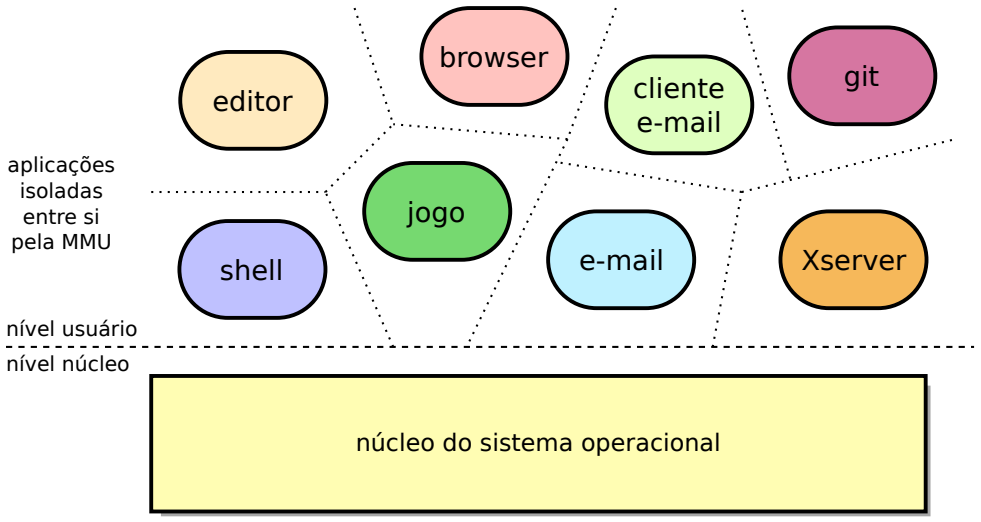

Figura 2.6: Separação entre o núcleo e as aplicações

2.3 Chamadas de sistema

O confinamento de cada aplicação em sua área de memória, imposto pela MMU aos acessos em memória 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 invocar, a partir da aplicação, as rotinas oferecidas pelo núcleo para o acesso ao

[2]E também irá gerar a famosa frase "este programa executou uma instrução ilegal e será finalizado", no caso do Windows.

hardware e demais serviços do SO? Deve-se lembrar que o código do núcleo reside em uma área de memória inacessível à aplicação, então operações como `jump` e `call` não funcionariam.

A resposta a esse problema está no mecanismo de interrupção, apresentado na Seção 2.2.2. Os processadores implementam instruções específicas para invocar serviços do núcleo, funcionando de forma similar às interrupções. Ao ser executada, essa instrução comuta o processador para o nível privilegiado e executa o código contido em uma rotina predefinida, como em uma interrupção. Por essa razão, esse mecanismo é denominado *interrupção de software*, ou *trap*.

A ativação de uma rotina do núcleo usando esse mecanismo 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 encerramento de tarefas, operadores de sincronização, 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 chamada e deporta devolve à aplicação os resultados obtidos.

Nos processadores modernos a chamada de sistema e seu retorno são feitos usando instruções específicas como `sysenter/sysexit` (x86 32 bits), ou `syscall/sysret` (x86 64 bits). Antes de invocar a chamada de sistema, alguns registradores do processador são preparados com valores específicos, como o número da operação desejada (*opcode*, geralmente no registrador AX), o endereço dos parâmetros da chamada, etc. As regras para o preenchimento dos registradores são específicas para cada chamada de sistema, em cada sistema operacional.

Um exemplo simples mas ilustrativo do uso de chamadas de sistema em ambiente Linux é apresentado a seguir. O programa em linguagem C imprime uma *string* na saída padrão (chamada `write`) e em seguida encerra a execução (chamada `exit`):

```
1 #include <unistd.h>
2
3 int main (int argc, char *argv[])
4 {
5   write (1, "Hello World!\n", 13) ; /* write string to stdout */
6   _exit (0) ;                      /* exit with no error */
7 }
```

Esse código em C, ao ser reescrito em Assembly x86, mostra claramente a preparação e invocação das chamadas de sistema:

```
; to assembly and link (in Linux 64 bit):
; nasm -f elf64 -o hello.o hello.asm; ld -o hello hello.o

section .data
msg db 'Hello World!', 0xA  ; output string (0xA = "\n")
len equ 13                  ; string size

section .text

global _start

_start:
   mov rax, 1               ; syscall opcode (1: write)
   mov rdi, 1               ; file descriptor (1: stdout)
   mov rsi, msg             ; data to write
   mov rdx, len             ; number of bytes to write
   syscall                  ; make syscall

   mov rax, 60              ; syscall opcode (60: _exit)
   mov rdi, 0               ; exit status (0: no error)
   syscall                  ; make syscall
```

A Figura 2.7 detalha o funcionamento básico da chamada de sistema `write`, que escreve dados em um arquivo previamente aberto). A seguinte sequência de passos é realizada:

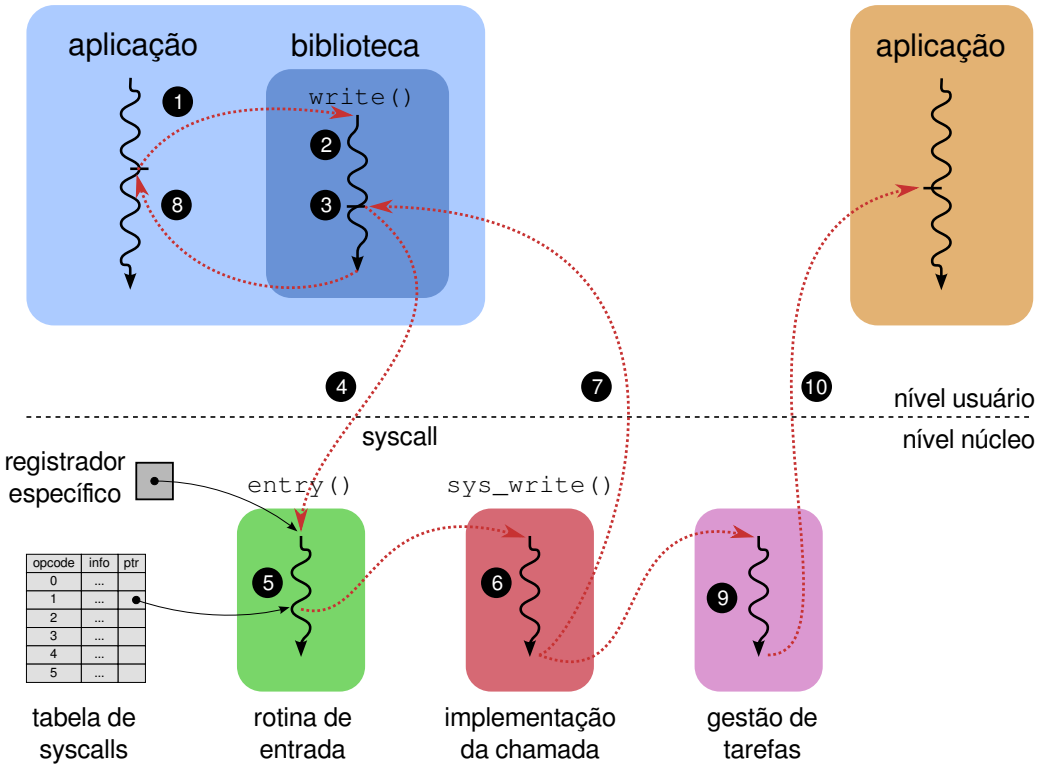

Figura 2.7: Roteiro típico de uma chamada de sistema

1. No nível usuário, a aplicação invoca a função `write(fd, &buffer, bytes)` da biblioteca de sistema (geralmente a biblioteca padrão da linguagem C).

2. A função `write` preenche os registradores da CPU com os parâmetros recebidos e escreve o *opcode* da chamada de sistema `write` no registrador AX.
3. A função `write` invoca uma chamada de sistema, através da instrução `syscall`.
4. O processador comuta para o nível privilegiado (*kernel level*) e transfere o controle para a rotina de entrada (`entry`), apontada por um registrador específico.
5. A rotina de entrada recebe em AX o *opcode* da operação desejada (1, `write`), consulta a tabela de chamadas de sistema mantida pelo núcleo e invoca a função que implementa essa operação dentro do núcleo (`sys_write`).
6. A função `sys_write` obtém o endereço dos parâmetros nos demais registradores, verifica a validade dos mesmos e efetua a operação desejada pela aplicação.
7. Ao final da execução da função, eventuais valores de retorno são escritos nos registradores (ou na área de memória da aplicação) e o processamento retorna à função `write`, em modo usuário.
8. A função `write` finaliza sua execução e retorna o controle ao código principal da aplicação.
9. Caso a operação solicitada não possa ser concluida imediatamente, a função `sys_write` passa o controle para a gerência de atividades, ao invés de retornar para a aplicação solicitante. Isto ocorre, por exemplo, quando é solicitada a leitura de uma entrada do teclado.
10. Na sequência, a gerência de atividades suspende a execução da aplicação solicitante e devolve o controle do processador a alguma outra aplicação que também esteja aguardando o retorno de uma chamada de sistema e cuja operação solicitada já tenha sido concluída.

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*) do 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.

O conjunto de chamadas de sistema de um SO pode ser dividido nas seguintes grandes áreas:

- Gestão de processos: criar, carregar código, terminar, esperar, ler/mudar atributos.
- Gestão da memória: alocar/liberar/modificar áreas de memória.
- Gestão de arquivos: criar, remover, abrir, fechar, ler, escrever, ler/mudar atributos.
- Comunicação: criar/destruir canais de comunicação, receber/enviar dados.
- Gestão de dispositivos: ler/mudar configurações, ler/escrever dados.
- Gestão do sistema: ler/mudar data e hora, desligar/suspender/reiniciar o sistema.

Exercícios

1. O que diferencia o *núcleo* do restante do sistema operacional?

2. Seria possível construir um sistema operacional seguro usando um processador que não tenha níveis de privilégio? Por quê?

3. Os processadores da família x86 possuem dois bits para definir o nível de privilégio, resultando em 4 níveis distintos. A maioria dos sistemas operacionais para esses processadores usam somente os níveis extremos (0 e 3, ou 00_2 e 11_2). Haveria alguma utilidade para os níveis intermediários?

4. Quais as diferenças entre *interrupções*, *exceções* e *traps*?

5. 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?

6. O comando em linguagem C `fopen` é uma chamada de sistema ou uma função de biblioteca? Por quê?

7. A operação em modo usuário permite ao processador executar somente parte das instruções disponíveis em seu conjunto de instruções. Quais das seguintes operações não deveriam ser permitidas em nível usuário? Por quê?

   (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

8. Considerando um processo em um sistema operacional com proteção de memória entre o núcleo e as aplicações, indique quais das seguintes ações do processo teriam de ser realizadas através de chamadas de sistema, justificando suas respostas:

   (a) Ler o relógio de tempo real do hardware.
   (b) Enviar um pacote através da rede.
   (c) Calcular uma exponenciação.
   (d) Preencher uma área de memória do processo com zeros.
   (e) Remover um arquivo do disco.

9. Coloque na ordem correta as ações abaixo, que ocorrem durante a execução da função `printf("Hello world")` por um processo (observe que nem todas as ações indicadas fazem parte da sequência).

(a) A rotina de tratamento da interrupção de software é ativada dentro do núcleo.

(b) A função `printf` finaliza sua execução e devolve o controle ao código do processo.

(c) A função de biblioteca `printf` recebe e processa os parâmetros de entrada (a string "`Hello world`").

(d) A função de biblioteca `printf` prepara os registradores para solicitar a chamada de sistema `write()`

(e) O disco rígido gera uma interrupção indicando a conclusão da operação.

(f) O escalonador escolhe o processo mais prioritário para execução.

(g) Uma interrupção de software é acionada.

(h) O processo chama a função `printf` da biblioteca C.

(i) A operação de escrita no terminal é efetuada ou agendada pela rotina de tratamento da interrupção.

(j) O controle volta para a função `printf` em modo usuário.

Atividades

1. O utilitário `strace` do UNIX permite observar a sequência de **chamadas de sistema** efetuadas por uma aplicação. Em um terminal, 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 sequência de **chamadas de biblioteca** efetuadas por uma aplicação. Em um terminal, 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 item anterior?

Referências

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

B. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc, 1994.

Google. *Android Platform Architecture*, April 2018. https://developer.android.com/guide/platform.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

W. Stallings. *Arquitetura e organização de computadores. 8ª ed*. Pearson, 2010.