Unidade 1: Introdução aos Sistemas Operacionais

Prof. Daniel Caetano

Objetivo: Apresentar a conceituação básica dos Sistemas Operacionais, suas principais funções e o histórico de sua evolução.

Bibliografia:

- MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.
- TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. São Paulo: Prentice Hall, 2003.

INTRODUÇÃO

* Problema: como tornar um computador útil?
  - Escolha de um sistema operacional
* Quais as tarefas básicas que devem ser oferecidas?
* Quando surgiram os sistemas operacionais?

Embora o uso de Sistemas Operacionais seja, hoje, parte do dia a dia da maioria das pessoas, muitas vezes sua função não é clara e é muito freqüente que algumas pessoas confundam as funções de alguns softwares aplicativos com a função do sistema operacional em si.

Neste aula será apresentado os conceitos fundamentais que definem "o que é" e "para que serve" um sistema operacional, além de apresentar sua evolução ao longo do tempo, em paralelo à evolução das máquinas e equipamentos.

1. O CONCEITO DE "SISTEMA OPERACIONAL"

* Função: executar ou auxiliar a execução de tarefas básicas
  - Ex: Carregar um programa, gerenciar impressão de documento
* Sistema operacional faz tudo?
* O que é?
  - Conjunto de rotinas, em geral de baixo nível
  - Carregador de Programas x Infinidade de Funções
  - Padronização de Acesso a Recursos x Compartilhamento de Recursos

Sempre que se deseja usar um equipamento, algumas tarefas precisam ser executadas: carregar um programa na memória, enviar alguns dados para a impressora ou simplesmente escrever algo na tela.

Estas estão entre algumas das funções para as quais um sistema operacional é projetado, seja para executá-las em sua completude, seja para facilitar a execução de algumas destas tarefas por parte do usuário ou de outros programas.

O nível de complexidade de um sistema operacional pode variar enormemente, assim como o nível de complexidade de utilização do mesmo. Isto significa que um sistema operacional pode fazer "mais" ou "menos" pelos usuários e pelos programas que nele irão ser executados.

Independentemente de sua complexidade, um sistema operacional não passa, entretanto, de um conjunto de rotinas executadas pelo processador, como qualquer outro programa que um usuário possa desenvolver. A diferença fundamental é que, no sistema operacional, estas rotinas são, em geral, rotinas de baixo nível, rotinas que conversam diretamente com o hardware.

Um sistema operacional pode ser tão simples quanto um mero carregador de programas (praticamente o que o DOS era) ou possuir uma infinidade de funções (a maioria dos sistemas operacionais atuais), mas praticamente todas estas funções podem ser definidas em duas categorias: **facilidade e padronização do acesso aos recursos de sistema** ou **compartilhamento de recursos do equipamento de forma organizada**.

1.1. Facilidade e Padronização do Acesso aos Recursos do Sistema

* Como facilitar o acesso a dispositivos?
    - Ex.: gravar um arquivo no HD
    - Como lidar com dispositivos de fabricantes diferentes?
* Virtualização de Dispositivos
    - Atuação como Intermediário
    - Ex.: Read / Write

Considerando que praticamente todo sistema computacional possui um grande número de dispositivos (drive de CD/DVD, monitores, impressoras, scanners etc) e que cada dispositivo tem uma operação bastante complexa, é interessante que exista uma maneira de poder utilizar tais dispositivos sem ter de se preocupar com o grande número de detalhes envolvidos no processo.

Por exemplo: o simples ato de escrever um arquivo em um *harddisk* exige um grande número de operações, envolvendo divisão do arquivo em blocos menores que caibam nos espaços livres disponíveis no disco e atualização das tabelas de diretório... para não falar em

todos os detalhes envolvidos com a escrita em si, como posicionalmento da cabeça de gravação no disco, escrita propriamente dita, verificação de erros etc.

Ora, a maioria destas atividades é um processo repetitivo e metódico que não requer qualquer intervenção do usuário. As únicas informações que realmente o usuário precisa fornecer são: o nome do arquivo, quais são os dados a gravar e em que disco ele deseja que estes dados sejam gravados. Todo o resto pode ser automatizado... e é exatamente o que o sistema operacional faz, neste caso.

Assim, o sistema operacional pode ser visto como uma interface entre o usuário/programa e o hardware, de forma a facilitar o acesso ao recursos do mesmo, como pode ser visto na figura 1.

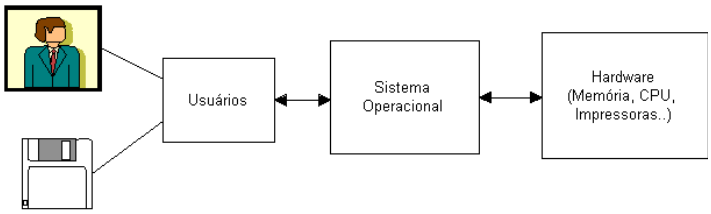


Figura 1 - Sistema Operacional como Interface entre usuários e recursos

Este tipo de "interface" que simplifica o acesso aos dispositivos (e os padroniza, em alguns casos) é também chamado de "**virtualização de dispositivos**". O caso mais comum deste tipo de virtualização são as operações "**read**" e "**write**" que, em geral, servem para ler e escrever em qualquer dispositivo, independente de seu funcionamento interno. Esta função de "virtualização" faz com que muitos usuários enxerguem o sistema operacional como uma "extensão da máquina" ou como uma *máquina extendida*.

É importante ressaltar, entretanto, que compiladores e bibliotecas fazem parte do Sistema Operacional. Embora sejam ferramentas muito importantes para o desenvolvimento, este tipo de software **não** faz parte do mesmo.

1.2. Compartilhamento de Recursos do Equipamento de Forma Organizada

* Compartilhar dispositivos?
* Vários programas tentando imprimir?
    a) Fazer o programa esperar
    b) Receber os dados e aguardar que a impressora esteja livre (spool)
* O que mais compartilhar?
    - Tela, teclado, mouse: múltiplas janelas
    - Disco, Rede?
* Múltiplos usuários
* Sistema Operacional: gerenciador de recursos!

Atualmente, o usuário médio está habituado a poder executar diversos programas ao mesmo tempo, em um mesmo equipamento. Ora, vários destes softwares podem precisar utilizar um mesmo dispositivo simultaneamente e, em geral, cada dispositivo só pode executar uma tarefa de cada vez. Como resolver este problema?

Mais uma vez surge uma das responsabilidades do sistema operacional, que é a de controlar o acesso a dispositivos de forma organizada, de maneira que os usuários deste sistema (no caso, softwares) possam compartilhar recursos. Assim, se um software está usando a impressora e um segundo deseja utilizá-la, o sistema operacional deve agir de uma de duas formas: **a) fazer com que o programa aguarde** ou **b) receber os dados a serem impressos e liberá-los para a impressora assim que ela esteja livre**. No caso de impressoras, é muito comum que a segunda alternativa seja implementada, e o nome deste recurso costuma ser *spool*.

Entretanto, não existem apenas equipamentos em que um usuário possui muitos programas. Em alguns sistemas, múltiplos usuários podem, ainda, estar usando o mesmo equipamento ao mesmo tempo. Neste caso, existe ainda mais a necessidade de um gerenciamento de recursos adequado, já que cada usuário terá seu conjunto de aplicativos, todos executando ao mesmo tempo, com os mesmos dispositivos, na mesma CPU e com a mesma memória.

Estas funções de gerenciamento fazem com que muitos usuários enxerguem o sistema operacional como um *gerenciador de recursos*.

1.3. Sistema Operacional como um dos Níveis de Máquina (OPCIONAL)

Em Arquitetura e Organização dos Computadores são, comumente, classificados os "níveis de máquina" de um computador, partindo do hardware até os aplicativos do usuário. Alguns autores subdividem a camada "Programas Aplicativos" em "Sistema Operacional" e "Aplicativos e Utilitários", estando a camada Sistema Operacional mais baixa na hierarquia, ou seja, mais próxima da máquina:

1. Programas Aplicativos
2. Sistema Operacional
3. Linguagem de Máquina
4. Micro Arquitetura
5. Dispositivos físicos

2. ELEMENTOS FUNDAMENTAIS DE UM SISTEMA OPERACIONAL

* Modo Supervisor x Modo Usuário
  - Ex.: No x86 => Supervisor: Anéis 0, 1 e 2; Usuário: Anel 3
  - Supervisor: todas operações acessíveis
  - Usuário: limitações no uso de memória e operações do processador
* O que roda em modo supervisor?
  - Normalmente uma parte do Sistema Operacional: Kernel (núcleo)
  - Função gerencial: o que o programa no modo usuário pode fazer
* Kernel = Sistema Operacional?
  - Não é consenso
  - Kernel (Supervisor) + Bibliotecas (Usuário) + Shell (Usuário)

Na maioria dos computadores atuais existem dois modos de operação: um deles é chamado "**modo supervisor**" e o outro "**modo de usuário**". No "modo supervisor" todas as instruções do processador e toda a memória estão acessíveis. Isso significa que um programa sendo executado neste modo tem acesso total à máquina e pode fazer com ela o que bem desejar. No "modo usuário", entretanto, o acesso é limitado tanto com relação às instruções quanto com relação à memória; apenas um subconjunto de instruções está disponível e apenas a região da memória designada àquele programa está acessível.

Mas quem (ou o que) define os limites dos programas sendo executados no "modo usuário"? Em geral, tais restrições são definidas por um programa que é executado no "modo supervisor" que, freqüentemente, é uma das partes mais importantes do Sistema Operacional: o **Kernel** (núcleo).

O Kernel é a parte mais importante do sistema operacional (e, segundo alguns autores como Tanenbaum (2003), a única parte do sistema operacional), pois é a parte que realiza as tarefas de baixo nível como a parte mais básica da abstração de hardware (faz com que todos os periféricos sejam "similares" para o restante do sistema) e cuida do gerenciamento de recursos básico (alocação de memória, alocação de processamento etc).

Entretanto, o Kernel não é a única parte do sistema operacional; é comum que o Kernel venha companhado de uma **Shell** (casca), que define sua interface com o usuário, e um conjunto de aplicativos básicos. Por se a parte com a qual o usuário interage, a Shell é aquilo que normalmente o usuário identifica como sendo o sistema operacional, embora ela seja apenas parte dele e, muitas vezes, possa ser trocada sem que se mude o sistema operacional (e é por esta razão que muitos não consideram a Shell como parte do Sistema Operacional).

Exemplos de Shell são o Command no DOS, o Command e o Explorer no Windows (não o Internet Explorer!), O Tracker no BeOS / Haiku, o Bash e o Xfree no Unix, o Presentation Manager e a WorkPlace Shell no OS/2 / eComStation e assim por diante.

3. EVOLUÇÃO DOS EQUIPAMENTOS E SISTEMAS OPERACIONAIS

* Divisão em 6 Fases
  - Também conhecidas como "Gerações"

A evolução dos equipamentos conhecidos hoje como "computadores" pode ser dividida em algumas fases. De certa forma, a evolução dos sistemas operacionais também acompanhou esta progressão em fases.

A seguir serão apresentadas algumas fases da evolução dos computadores e, em conjunto, a evolução dos sistemas operacionais.

3.1. Primeira Fase

* 1940 a 1955
* ENIAC: 5000 adições por segundo
  - O que isso representa hoje?
* Linguagem de Máquina: Wire-up
* Não existia conceito de Sistema Operacional
* Falta de Confiabilidade: executar várias vezes!
* EDVAC, IAS, UNIVAC I...

A primeira fase (ou primeira geração) dos computadores ocorreu aproximadamente durante o período entre 1940 e 1955, quando surgiram os primeiros computadores digitais usados durante a segunda guerra mundial.

O ENIAC (Eletronic Numerical Integrator and Computer) foi o primeiro computador de propósito geral, era gigantesco e tinha uma capacidade de processamento em torno de 5000 adições por segundo, valor este muito inferior ao de uma calculadora atual da Hewlett-Packard.

Estes primeiros computadores eram programados em grande parte com o método chamado *wire-up*, isto é, conectando fisicamente, com fios, diversos polos, fechando bits de forma a construir um programa. A programação era, assim, feita diretamente em linguagem de máquina e o equipamento não oferecia qualquer recurso para facilitar este trabalho de programação. Não existia, assim, o conceito de Sistema Operacional nesta fase.

Outros grandes computadores construídos na época incluem o EDVAC, o IAS e o UNIVAC I, construído pela IBM para a computação do censo americano de 1950.

3.2. Segunda Fase

* 1955 a 1965
* Transístores => confiabilidade
* Memórias magnéticas
    - Maior velocidade e maior capacidade
* Programas armazenados em memória (Modelo de Von Neumann)
    - Funções de E/S => embrião dos Sistemas Operacionais
        + Eliminação de bugs
        + Interface padronizada para dispositivos de E/S
        + Escrita com Independência de Dispositivos
* Automação de Processo Sequenciais
    - Processamento Batch => vários programas em sequência
        + Cartão x Batch: qual a vantagem?
    - Sem intervenção
* Escrita direta entre dispositivos (DMA)

A segunda fase ocorreu aproximadamente entre 1955 e 1965, e a grande inovação era o uso de transístores, o que permitiu uma grande redução no tamanho dos equipamentos e aumento de sua velocidade, além do aumento da **confiabilidade** do processamento. Também é desta época o surgimento das memórias magnéticas, que permitiram um aumento na capacidade e velocidade do armazenamento.

Nesta fase surgiram as primeiras linguagens e compiladores e, pela primeira vez, surgiu o conceito de sistema operacional como um software para automatizar todas as tarefas repetitivas utilizadas por diversos softwares e que, até então, eram realizadas manualmente.

O primeiro grande grupo de funções que acabou sendo incluído ao Sistema Operacional foi o grupo de operações de entrada e saída (E/S ou I/O). Isso permitiu duas grandes mudanças: a primeira foi que os programadores deixaram de precisar escrever suas próprias rotinas de escrita e leitura, o que facilitava muito o desenvolvimento e reduzia muito o número de defeitos. A segunda foi que convencionou-se implementar todas as operações de I/O do sistema operacional com a mesma *interface*, isto é, independente da escrita ser em um disco, fita ou mesmo rede e impressora, o "comando" a ser usado no sistema operacional para executar a tarefa era o mesmo. Este foi o surgimento do conceito de *independência de dispositivos*.

Um outro avanço desta geração foi a automação de processamentos seqüenciais (processamento *batch*). Originalmente, sempre que se desejasse executar um programa, o programador deveria inserir este programa no equipamento (através de um cartão perfurado), este programa seria executado e finalmente o resultado seria impresso. Entretanto, em geral este processamento durava horas e era comum que se passassem horas até alguém perceber que o processamento havia finalizado. Isso fazia com que o equipamento ficasse ocioso

mesmo que muitas tarefas ainda estivessem por ser executadas, já que o equipamento dependia que um ser humano fosse até ele e o alimentasse com um novo programa.

Nesta geração, então, passou a ser possível introduzir diversos programas e o "sistema operacional" existente era capaz de executá-los em seqüência, ou seja, assim que um terminava, ele iniciava o seguinte e assim por diante, eliminando o problema da ociosidade. Ainda no final desta fase a IBM criou o conceito de "canal", hoje comumente chamado de "DMA", que permitia a escrita direta entre dispositivos sem a necessidade de intervenção da CPU.

3.3. Terceira Fase

* 1965 a 1980
* Circuitos Integrados => redução de tamanho e custos
* IBM Série 360 / DEC PDP-8
* ...Sistemas Operacionais... Ex.: OS/360
    - Multiprogramação: multitarefa cooperativa
        + Processamento de um programa durante espera de outro
    - Melhora no processamento batch
        + Prioridades
        + Spooling => usada até hoje na impressão
* Terminais de Impressão e de Vídeo
    - Interação Online
* PDP-7 (POSIX/UNIX e Linguagem C), Computadores Apple, CP/M

Esta fase ocorreu mais ou menos no período de 1965 a 1980, e foi marcada pela utilização dos circuitos integrados, que reduziram tamanho e custos e ampliaram enormemente a capacidade de armazenamento, processamento e confiabilidade dos computadores.

Nesta época surgiu o conceito de família de processadores (IBM Série 360), em que vários equipamentos, com dispositivos diferentes conectados, eram compatíveis entre si. Também surgiram computadores de custo menor, como o PDP-8 da DEC.

Com o grande aumento de recursos destes equipamentos, os novos sistemas operacionais (como o OS/360) traziam novas possibilidades de gerência de processamento, permitindo que enquanto quando um programa esperava pela entrada de dados do usuário, outro fosse processado. Esta tecnologia ficou conhecida como *multiprogramação* e é uma técnica básica envolvida na *multitarefa cooperativa*.

O processamento "*batch*" também melhorou, permitindo troca da ordem dos processos a serem executados de acordo com a prioridade. Esta técnica ficou mais tarde conhecida como "*spooling*" e é usada até hoje, por exemplo, para o gerenciamento de impressão. Nesta geração também passou a existir uma "interação online"; foram criados os

primeiros terminais de vídeo e teclados para comunicação com o software *durante sua execução*.

Ainda nesta geração, a multiprogramação evoluiu de maneira a melhorar os tempos de resposta na interação com os usuários, desemvolvendo o conceito de *time-sharing*, isto é, cada processo compartilha a CPU por um intervalo de tempo. Este conceito é a base da *multitarefa preemptiva*.

Surgiu nesta fase, ainda, o sistema operacional UNIX, concebido inicialmente para o computador PDP-7, desenvolvido em Linguagem C, e tornou-se bastante conhecido por sua portabilidade. Outras grandes novidades desta época foram os computadores de 8 bits da Apple e o sistema operacional CP/M (Control Program Monitor).

Vale ressaltar que, nesta geração, houve a criação do padrão POSIX (Portable Operating System IX), que definiu uma interface mínima que sistemas UNIX devem suportar. Muitos sistemas operacionais atuais suportam o padrão POSIX, tanto os derivados do UNIX (Linux, por exemplo) quanto não derivados (por exemplo, MacOS X, BeOS/Haiku e, em certo nível, OS/2 / eComStation). O suporte ao padrão POSIX em geral significa uma maior facilidade de portar programas de uma destas plataformas para outra.

3.4. Quarta Fase

* 1980 a 1990
* Integração em Larga Escala (LSI e VLSI)
* Computadores Pessoais (no Brasil, do MSX ao IBM-PC)
    - Recursos limitados: DOS era suficiente (sem spooling, time-sharing etc)
* Computadores de Grande Porte
    - VMS: multitarefa monousuário
* Computadores Multiprocessados
* LANs, WANs, TCP/IP... Sistemas Operacionais de Rede
* Primeiras implementações das GUIs

Nesta fase, que durou toda a década de 1980, a integração em larga escala (LSI e VLSI) permitiram uma redução substancial no tamanho e no preço dos equipamentos. Com isso houve o surgimento de diversos computadores menores mas muito potentes (variando desde os mais simples como o MSX até os mais poderosos IBM-PCs), ambiente no qual surgiu o DOS (Disk Operating System), base dos "computadores pessoais" do período. Estes equipamentos tinham processamento relativamente limitado e, portanto, o DOS não suportava muitas das características de multiprogramação, spooling, time-sharing e outros.

No campo dos computadores de grande porte, surgiu o sistema VMS (Virtual Machine System) que, implementando todos os recursos concebidos até então, criou oficialmente o conceito de *multitarefa* em um sistema monousuário.

Nesta fase surgiram os computadores capazes de *multiprocessamento*, com várias CPUs em paralelo, e os primeiros sistemas capazes de lidar com este tipo de característica também surgiram. Nesta fase houve proliferação das LANs e WANs, com o surgimento de diversos protocolos de comunicação e uma grande aceitação do protocolo TCP/IP. Muitos sistemas operacionais passaram a incluir controles para esta comunicação e passaram a ser chamados de *sistemas operacionais de rede*.

A preocupação com a facilidade de uso, iniciada nos laboratórios na fase anterior, tomou corpo em computadores da Apple, como o Lisa e, posteriormente no computador MacIntosh, com o sistema operacional MacOS.

Alguns autores (como Tanenbaum, 2003) não consideram a quinta e sextas fases, colocando os avanços posteriores ao da quarta fase dentro da própria quarta fase. Por questões didáticas, neste trabalho foi feita a opção pela separação.

3.5. Quinta Fase

* 1990 a 2000
* Enorme aumento da capacidade de processamento e armazenamento
* IA, Banco de Dados, multimídia...
* Multitarefa nos computadores pessoais
  - Linux, Windows NT, OS/2
* GUIs se tornaram padrão

A quinta fase compreendeu basicamente a década de 1990, sendo a tônica principal o aumento da capacidade de processamento e armazenamento em proporções não previstas anteriormente, possibilitando aplicação de inteligência artificial, bancos de dados e mutlimídia em praticamente qualquer tipo de aplicação, tornando-as muito mais complexas.

Nesta fase os computadores se tornaram muito baratos e passaram a fazer parte da vida de praticamente todas as pessoas. O conceito de *processamento distribuído* passou a fazer parte das pesquisas e a *multitarefa* veio para os computadores pessoais, em sistemas operacionais como Linux, IBM OS/2 e Microsoft Windows NT.

A preocupação com a facilidade de uso dominou o mercado dos computadores e grandes inovações em termos de interface com o usuário surgiram, nos mais diversos paradigmas, por todos os sistemas operacionais existentes.

3.6. Sexta Fase

* 2000 até hoje
* Rede sem fio ubíqua
* Limite físico para processamento de uma CPU
  - Multiprocessamento nos computadores pessoais
* Processamento Distribuído é comum
* Computação móvel

A sexta fase teve início juntamente com o século XXI e ainda não foi finalizada. As inovações trazidas são conhecidas pela maioria das pessoas, como a ubiquidade do acesso à rede, com redes sem fio e internet, com um aparente limite físico estabelecido da capacidade de processamento de uma unidade central de processamento e o *multiprocessamento* chegando aos computadores pessoais a baixos preços.

A quantidade de memória e velocidade de comunicação das redes permitem que grandes massas de dados sejam processadas e transmitidas, possibilitando video-conferências a um baixo custo. O processamento distribuído tornou-se uma realidade comum, embora ainda explorada apenas por aplicações científicas.

A computação móvel tornou-se uma realidade, com a proliferação dos laptops e palmtops, levando os recursos computacionais a qualquer lugar.

Os sistmas operacionais da quinta fase evoluíram para atender às novas necessidades e suportar a todos estes novos recursos.

4. TIPOS COMUNS DE SISTEMA OPERACIONAL

* Grande número de aplicações
* Sistemas especializados

O histórico da evolução dos equipamentos computacionais e seus diferentes usos levaram a um grande número de sistemas operacionais, especializados em diferentes usos. Nesta seção serão vistos alguns deles.

4.1. Sistemas Operacionais para Computadores de Grande Porte

* Capacidade de armazenamento massiva
* Dezenas de processadores em paralelo
* Servidores de rede de grande demanda
  - B2B
* Ex.: OS/390, OS/400

Computadores de grande porte são, essencialmente, computadores com grande capacidade de armazenamento, com centenas ou milhares de discos e vários terabytes de capacidade de armazenamento. Seu uso nos dias atuais é basicamente em servidores web de grande demanda (como comércio eletrônico) e aplicações *business-to-business*.

Os sistemas deste tipo de computador normalmente oferecem processamento em *batch*, processamento de transações e tempo compartilhado. Neste caso, processamento de transções significa o processamento de grandes quantidades de pequenas requisições. Um sistema desta categoria é o OS/390, descentente do OS/360.

4.2. Sistemas Operacionais de Servidores

* Grande capacidade de armazenamento
* Várias unidades de processamento
* Vários usuários simultâneos
* Compartilhamento de recursos
  - Impressão
  - Publicação Web
  - E-mail
* Ex.: variantes de Unix, Windows Server etc.

Sistemas Operacionais de Servidores podem ser executados em computadores servidores, que são computadores pessoais muito potentes, podem rodar em estações de trabalho ou mesmo em computadores de grande porte. Estes sistemas são feitos para trabalhar com um grande número de usuários simultâneos e compartilhar recursos de hardware e software. São usados servidores para impressão, para publicação de páginas web e manutenção de contas de e-mail, por exemplo.

Alguns exemplos de sistemas do tipo Servidores são variantes de Unix (como o Linux), Windows Server etc.

4.3. Sistemas Operacionais Multiprocessadores

* Capazes de utilizar vários processadores
* Praticamente todos os SOs de computadores pressoais atuais

Sistemas Operacionais Multiprocessadores são aqueles adaptados para tirar proveito da existência de múltiplos processadores em um mesmo equipamento. Praticamente a totalidade dos sistemas operacionais atuais consegue tirar proveito destes recursos.

4.4. Sistemas Operacionais de Computadores Pessoais

* Interface amigável com o usuário
* Capazes de gerenciar os dispositivos mais comuns do dia-a-dia
* Ex.: Windows 7, Linux, MacOS X

Sistemas Operacionais de Computadores Pessoais são aqueles voltados a computadores pessoais, com o objetivo de fornecer uma boa interface de operação para um único usuário. O número de opções nesta área é imenso, variando dos mais comuns Windows XP / Vista / 7, Linux e MacOS X até os menos comuns como Haiku e eComStation, e os mais antigos e descontinuados Windows 98, MacOS, BeOS, OS/2, dentre outros.

4.5. Sistemas Operacionais de Tempo Real

* Parâmetro Fundamental: tempo de resposta
* Controle de equipamentos e atividades críticas
  - Fissão atômica, prensas etc.
* Tempo Real Crítico x Tempo Real Não Crítico
* Ex.: VxWorks, QNX, LinuxRT...

Sistemas Operacionais de Tempo Real são aqueles em que o tempo de resposta é um parâmetro fundamental. Normalmente são sistemas operacionais que controlam processos industriais ou experimentos críticos (como controle térmico de fissão atômica). Existem duas categorias básicias: os **sistemas de tempo real crítico**, em que o menor atraso no tempo de resposta pode ocasionar grandes problemas e **sistemas de tempo real não crítico**, onde um pequeno atraso pode ser tolerado.

O número de sistemas operacionais de tempo real não é muito amplo. VxWorks e QNX são os mais conhecidos, embora exista a menos conhecida variante do Linux, o LinuxRT, que é adaptado para trabalho em tempo real. O antigo OS/2 e o atual eComStation permitem configuração para trabalho próximo às exigências de tempo real, mas não podem

ser considerados nesta categoria (embora tenham sido, historicamente, usados para controle de usinas nucleares).

4.6. Sistemas Operacionais Embarcados

* Voltados a PDAs, videogames, celulares...
* Restrições de memória e processamento
* Interfaces específicas
* Ex.: Google Android, Nokia Symbian, Windows CE...

Sistemas Operacionais Embarcados são aqueles que permitem o funcionamento de dispositivos gerais, normalmente portáteis, como PDAs, livros eletrônicos etc. Entretanto, eles também podem estar presentes em monitores, TVs, fornos de micro-ondas etc.

Alguns destes sistemas têm requisitos de tempo real, mas normalmente apresentam grandes restrições de tamanho, uso de memória e consumo de energia, o que faz com que possuam uma categoria especial. Os sistemas mais conhecidos desta categoria são Android, Wndows CE e Symbian (Nokia), além do descontinuado PalmOS.

4.7. Sistemas Operacionais de Cartões Inteligentes

* Menores existentes
* Restrições severas de consumo de memória e energia
* Alguns capazes de executar aplicações J2ME

Sistemas Operacionais de Cartões Inteligentes são os menores existentes, possuindo severíssimas restrições de memória e consumo de energia. São comumente sistemas proprietários e suas funções variam enormemente. Alguns cartões deste tipo são capazes até mesmo de executar pequenas aplicações Java (especialmente desenvolvidas, no padrão J2ME).

5. BIBLIOGRAFIA

DAVIS, W.S. Sistemas Operacionais: uma visão sistêmica. São Paulo: Ed. Campus, 1991.
MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.
SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.
TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. São Paulo: Prentice Hall, 2003.

Unidade 2: Arquitetura dos Sistemas Operacionais

Prof. Daniel Caetano

Objetivo: Apresentar os elementos básicos de um sistema operacional, os conceitos das arquiteturas monolítica e microkernel e os modelos em camadas.

Bibliografia:

- MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.
- TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. São Paulo: Prentice Hall, 2003.
- SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.

INTRODUÇÃO

* Problema: como entender sistema operacional?
  - Complexo => conjunto de funções de suporte
* Quais são os Componentes do SO
  - Modo Supervisor x Modo Usuário

Em geral, a compreensão da estrutura de um sistema operacional é complexa, mas esta complexidade vem de um fato já citado: o sistema operacional não é um programa tradicional, que começa e segue até o fim de sua execução, mas sim é composto de um conjunto de funções que são executadas em determinadas condições.

Algumas partes destas funções são executadas no "modo supervisor" (em geral, são as que fazem parte do núcleo e/ou drivers), mas outras são executadas no "modo usuário". Quais são os componentes que são executados em cada um dos modos variam bastante, mas os componentes existentes são muito semelhantes em todos os sistemas operacionais.

Será inicialmente apresentada uma visão geral destes componentes, alguns exemplos de como este componentes podem ser agrupados nas duas arquiteturas mais comuns de sistema operacional, sendo também apresentados alguns modelos em camadas e, finalmente, como funcionam as chamadas de sistema.

1. COMPONENTES DE UM SISTEMA OPERACIONAL

* Divisão de Estudo
* Gerenciamento de Processos
    - processo: "programa em execução"
        + spooling é processo!
    - Ocupam: memória, e/s, cpu...
        + Subprocessos?
    - Funções Relativas a Processos
        + Criação/Remoção/Suspensão/Reativação/Sincronia/Comunicação
* Gerenciamento de Memória Principal
    - Cada processo => memória só dele (Endereçamento Virtual)
    - Controle DMA
    - Funções Relativas a Memória
        + Informação de Uso
        + Alocar/Remover (Dados e Processos)
        + O que fazer quando memória está disponível
* Gerenciamento de E/S
    - Controlar dispositivos diversos (Entrada/Saída)
        + Cada dispositivo é diferente!
    - Gerenciar memória => armazenamento temporário (buffers)
    - Interface genérica para controladores
    - Rotinas de Controle Individual (drivers)
* Gerenciamento de Arquivos
    - Memória Principal => em geral, volátil
    - Armazenamento Permanente de Programas e Dados
    - Conceito de Arquivo
    - Funções Relativas a Arquivos
        + Criação/Remoção/Manipulação/Diretórios
        + Memória Secundária
            = Out-of-memory?
            = Memória Virtual
            = Paginação (Swap)
            = Funções: Ger. Espaço/Alocação/Ordenação
* Processamento Distribuído
    - Máquinas diferentes operando como uma
    - Distribuído x Paralelo
* Interpretador de Comando
    - Interface com Usuário
    - Textual x Gráfica
    - Funções
        + Gerenciamento do Hardware
        + Carregar Programas / Finalizar Processos
        + Acesso ao sistema de arquivos

Independente do sistema operacional, alguns componentes estão sempre presentes em todos eles. Na prática, algumas vezes alguns destes componentes não são totalmente separados, mas esta separação é feita para facilitar o estudo. Os componentes básicos de um sistema operacional são:

Gerenciamento de Processos: simplificadamente, pode-se definir *processo* como sendo um **programa em execução**. Entretanto, o conceito de processo é mais genérico, de forma que um serviço qualquer, como o *spooling* de impressão, também pode ser considerado um processo. Os processos ocupam memória, utilizam entrada e saída, ocupam o processador... e, inclusive, podem possuir *subprocessos* a serem executados simultaneamente.

O sistema operacional precisa permitir que tudo isso funcione em harmonia e, para tal, tem as seguintes funções diante relativas ao gerenciamento de processos:

a) Criação e remoção de processos (de sistema ou do usuário).
b) Suspensão e reativação de processos.
c) Sincronização de processos.
d) Comunicação entre processos.
e) Tratamento de impasses entre processos.

Gerenciamento da Memória Principal: A memória principal é um dos principais recursos de um computador, sendo o local onde processos e seus dados ficam armazenados. Os processos costumam ser executados de maneira que, para cada processo, é como se o equipamento (e a memória) fosse somente sua. Isto significa que todos os processos podem pensar que estão sendo executados no mesmo local da memória. Isto certamente não será verdade, mas os processadores modernos possuem recursos para permitir que os processos sejam executados com essa ilusão. Este é um recurso chamado de *espaço de endereçamento virtual*.

Além dos processos sendo executados pela CPU, também os dispositivos acessam a memória através de mecanismos de DMA. Assim, o sistema operacional também precisa fornecer algumas funções relativas ao gerenciamento de memória. Algumas delas são:

a) Manter a informação de quais partes da memória estão em uso (e por quem).
b) Decidir quais processos devem ser carregados quando a memória ficar disponível.
c) Alocar e remover processos e dados da memória quando necessário.
d) Controlar o sistema de enedereçamento virtual dos processos.

Gerenciamento de Sistema de E/S: Em algum momento, todo processo precisa se comunicar com o mundo exterior, seja para obter informações, seja para fornecer informações. Estas comunicações são realizadas pelos processos de Entrada e Saída (E/S ou, do inglês, Input/Output, I/O). Assim, se um processo deseja imprimir algo na tela ou na impressora ou mesmo se o gerenciador de arquivos do sistema precisa escrever um byte no disco, será necessária uma operação de E/S. É interessante que as particularidades de cada dispositivo de E/S sejam escondidas do usuário e dos processos, tornando-os tão similares quanto possível.

Para realizar esta tarefa, existe o gerenciamento de E/S, que tem as seguintes componentes:

a) Um componente de gerenciamento de memória, que controla os usos das áreas de armazenamento temporário.
b) Uma interface geral para os diversos controladores de dispositivo.
c) Rotinas de controle individuais para cada dispositivo (drivers).

Gerenciamento de Arquivos: Embora os programas só realizem trabalho útil enquanto estão sendo executados, ou seja, na forma de *processos*, para que estes programas possam ser executados é necessário que eles estejam armazenados em algum lugar. Também os dados necessários à execução de um programa, bem como aqueles gerados pelo mesmo, precisam ser armazenados. Isto normalmente não é feito na memória principal, dado o fato que, em geral, ela é composta de memória volátil.

Assim, os computadores usam uma infinidade de dispositivos de armazenamento para guardar programas e informações, dispositivos estes que contém memória permanente, como harddisks, pen-drives, disquetes, dentre outros. Independente do dispositivo, é necessário que as informações armazenadas possam ser recuperadas facilmente e, de preferência, de maneira similar do ponto de vista do usuário e dos programas. Isto é conseguido através do conceito de *arquivo* que é o nome dado a um conjunto coeso de informações armazenadas no dispositivo de memória permanente. Assim, o sistema operacional também precisa fornecer algumas funções relativas ao gerenciamento destes arquivos. Algumas delas são:

a) Criação e remoção de arquivos (e diretórios).
b) Suporte a primitivas de manipulação de arquivos e diretórios.
c) Mapeamento de arquivos em memória secundária.
d) Cópia de arquivos em meios de armazenamento permanentes (não-voláteis).

Gerenciamento de Memória Secundária: Como já comentado no sistema de arquivos, um programa só produz trabalho útil quando está na forma de processo, ou seja, na memória principal. Entretanto, esta memória principal é usualmente pequena para que todos os programas fiquem armazenados nela - além do fato que quando o computador é desligado, os dados são perdidos. Para resolver uma parte do problema, existe o sistema de arquivos, que trabalha em conjunto com o gerenciamento de E/S e gerenciamento de memória secundária para atingir seus objetivos, permitindo que programas que não estão sendo usados sejam armazenados em uma unidade de memória secundária (não-volátil).

Por outro lado, há momentos em que a memória principal é pequena demais até mesmo para armazenar todos os processos atuais; neste caso, o sistema operacional lança mão de um recurso chamado *memória virtual* e, através de um arquivo especial chamado *arquivo de paginação* (ou arquivo de *swap*), permite que alguns processos e seus dados sejam realocados para a memória secundária nos momentos em que estiverem inativos, liberando espaço na memória principal, usando assim a memória secundária como uma real extensão da memória principal.

Dentre as principais funções do gerenciamento de memória secundária estão:

a) Gerenciamento de espaço livre.
b) Alocação de espaço na memória.
c) Ordenamento e seleção das operações para o uso de discos.

Em geral o sistema de arquivos trabalha solicita serviços do gerenciamento de memória secundária que, por sua vez, solicita serviços de E/S ao gerenciador de E/S.

Processamento Distribuído: Dependendo do sistema operacional e do hardware em que ele trabalha, outros componentes podem ser necessários. Um deles é aquele que permite o *processamento distribuído*, isto é, em que processos podem estar sendo executados em diferentes máquinas (conjunto de processador, memória e dispositivos de E/S) e se comunicarem como se estivessem em uma mesma máquina.

Interpretadores de Comandos: Os interpretadores de comandos são uma parte bastante importante do sistema, sendo através dela que o usuário se comunica com o sistema. Os interpretadores de comando podem ser textuais (prompt de comando) ou gráficos (como na maioria dos sistemas atuais). Algumas das funções dos interpretadores de comandos são:

a) Permitir gerenciamento do hardware.
b) Carregar programas e finalizar processos.
c) Permitir acesso ao sistema de arquivos.

2. ORGANIZAÇÃO DE UM SISTEMA OPERACIONAL

* Componentes: Modo Supervisor ou Modo Usuário?
* Kernel (Núcleo)
    - Parte que "roda" em modo supervisor
        + Monolítico x Microkernel
* Componentes de Modo Usuário
    - *System call*
    - Desempenho da *System Call*
        + Monolítico x uKernel => Velocidade x Portabilidade

Como dito anteriormente, os componentes de um sistema operacional podem estar agrupados de diferentes maneiras, de modo que alguns componentes podem ser executados em *modo supervisor* ou em *modo usuário*, sendo esta uma definição diferente em cada sistema.

Em geral, chama-se de *kernel* a parte principal do sistema operacional, aquela que é executada no *modo supervisor*. Os componentes que não fizerem parte do *kernel* serão

executados no *modo usuário*. Quando a maior parte dos componentes estão incluídas no *kernel*, isto é, sendo executadas no modo supervisor, dá-se o nome de **sistema monolítico**, pois todos os componentes estão agrupados em um grande bloco. Quando a maior parte dos componentes estão fora do kernel, dá-se o nome de **sistema microkernel** (uKernel), já que, neste caso, o kernel será bem menor.

Quando um componente que está sendo executado no *modo usuário* precisa de alguma função do *kernel*, ele executa uma *chamada de systema* ou, em inglês, uma *system call*. Uma chamada de sistema não é, em geral, um processo complexo, mas pode ser uma ação lenta (proporcionalmente, considerando as tarefas a serem realizadas e o número de vezes que elas ocorrem. Isto faz com que os sistemas microkernel sejam ligeiramente mais pesados (lentos) que os sistemas monolíticos. Por outro lado, a modularização conseguida pelo modelo microkernel permite que o sistema seja expandido mais facilmente e que seja portado para outros equipamentos e plataformas com uma facilidade muito maior.

A regra é que, em geral, quanto mais direto for o acesso do software/processo ao hardware, mais leve (rápido) será a execução do mesmo; por outro lado, o acesso direto limita a compatibilidade, fazendo com que o próprio software seja responsável por lidar com diferentes tipos de dispositvos e ambientes. Como isso normalmente não acontece, quanto mais direto for o acesso de um software a um hardware, menos portável e menos compatível ele é.

2.1. Modelo em Camadas

* Modelo MS-DOS: Monolítico
* Modelo UNIX: Monolítico
  - Mach: Micro Kernel
* Modelo OS/2: Monolítico
* Modelo NT: Híbrido
* Modelo Vista: "quase" microkernel

Normalmente é possível definir um sistema operacional como um conjunto de camadas. Idealmente, uma camada só pode se comunicar diretamente com as camadas que fazem fronteira com a mesma. Em alguns casos, entretanto, há uma "passagem direta" entre camadas, permitindo que uma camada superior acesse camadas inferiores. Como essa "passagem direta" elimina intermediários (camadas intermediárias), há um evidente ganho de performance; por outro lado, fazendo isso caminha-se em direção à redução de portabilidade e compatibilidade.

A idéia é que, à medida em que se caminha da camada mais baixa (hardware) para a mais alta (aplicativos) o *nível de abstração de hardware* aumente. Assim, normalmente a camada logo acima do hardware é a camada de *controladores de dispositivo*, passando pelo *kernel* do sistema (em suas possíveis diversas camadas e subsistemas), *bibliotecas de sistema* e, finalmente, *aplicativos do usuário*.

2.1.1. Modelo em Camadas do MS-DOS

O MS-DOS é um sistema operacional bastante simples, que não tem uma estrutura em camadas muito bem definida, pois ele começou como um sistema pequeno, sem grandes pretenções em termos de extensões futuras. O único objetivo de seu projeto foi permitir o acesso ao maior número de recursos ocupando o menor espaço possível e, por esta razão, seus componentes e módulos não são muito bem definidos.

Por exemplo: no MS-DOS os aplicativos podem ter acesso direto aos dispositivos, o que dificulta muito a implementação de multi-tarefa, além de possibilitar travamentos no sistema através de erros simples de programação. Ele é classificado como um sistema **monolítico**, mas não é possível separar claramente os elementos do kernel. A figura 1 apresenta um possível (e aproximado) modelo em camadas para o MS-DOS (SILBERSCHATZ; GALVIN, 2000):

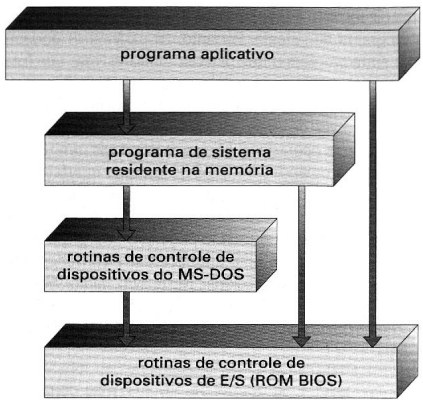


Figura 1: Organização em camadas do MS-DOS

2.1.2. Modelo em Camadas do UNIX

O UNIX é um outro sistema que foi fortemente influenciado por limitações de hardware e não foi projetado com uma estrutura bem definida. O UNIX padrão é um sistema **monolítico**, contendo dois blocos básicos: o núcleo (kernel) e os programas de sistema. Praticamente todos os componentes (exceção feita aos interpretadores de comandos e os comandos em si) estão no kernel. Um modelo simplificado para as camads do Unix está representado na figura 2a:

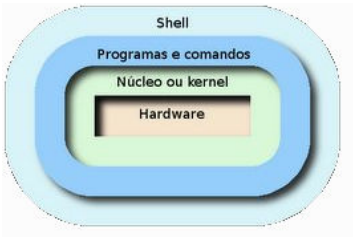


Figura 2a: Organização em camadas simplificada do Unix
(fonte: http://br.gnome.org/bin/view/TWikiBar/TWikiBarPapo001)

Este mesmo modelo, mais detalhado, é apresentado na figura 2b (SILBERSCHATZ; GALVIN, 2000):

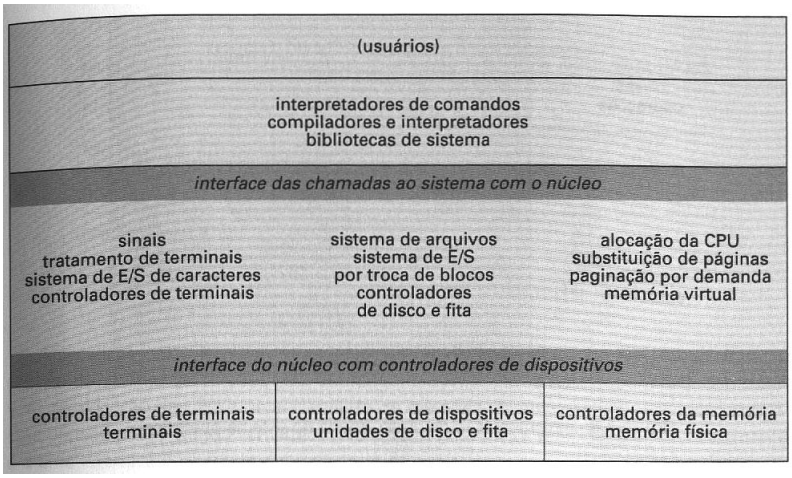


Figura 2b: Organização em camadas do UNIX

Nem toda implementação que segue as linhas do UNIX é assim. O IBM **AIX** (a implementação UNIX da IBM) possui mais camadas, em uma estrutura mais organizada, embora ainda seja **monolítico**.

Um outro sistema, implementado na universidade de Carnegie Mellon, chama-se **Mach** e, embora não seja exatamente um UNIX, tem muitas características similares e é bem melhor estruturado, sendo inclusive um exemplo de implementação do tipo **microkernel**.

2.1.3. Modelo em Camadas do OS/2 / eComStation

O IBM OS/2 é um ótimo exemplo de arquitetura que segue um modelo em camadas projetado com o objetivo de proporcionar uma adequada abstração de hardware. Existe o nível mais baixo, onde se situam os **controladores de dispositivo**, num nível mais acima existe o **núcleo de sistema**, com os gerenciadores de dispositivos, memória e processos, na camada acima há os diversos **subsistemas** (subsistema de arquivos, subsistema de áudio, subsistema de vídeo, subsistema de impressão, subsistema de redes...).

Acima dessa camada de subsistemas, existe a camada das **bibliotecas de sistema**, que oferecem acesso organizado a todos os subsistemas. Acima disso tudo é que são executados os aplicativos do usuário. Esta arquitetura pode ser vista na figura 3 (SILBERSCHATZ; GALVIN, 2000):

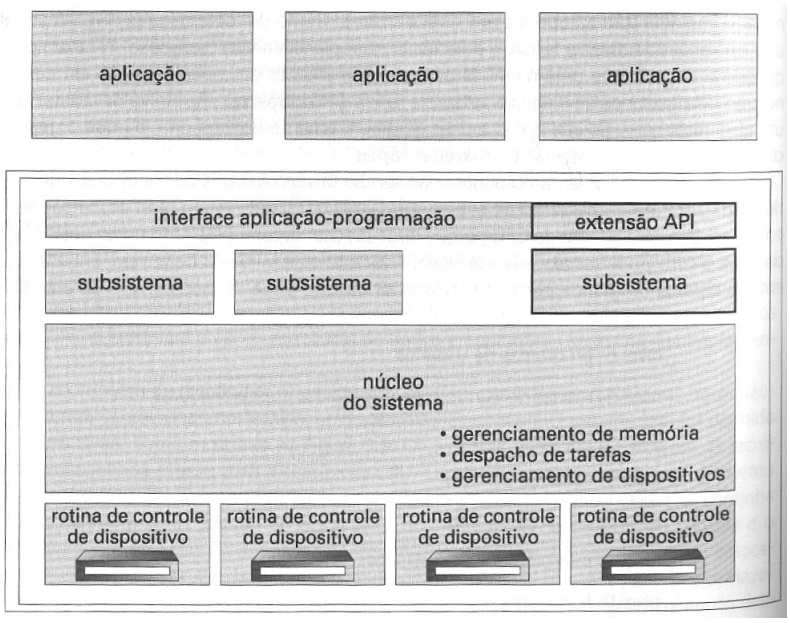


Figura 3: Organização em camadas do eComStation (OS/2)

Esta organização - que é originalmente monolítica - se mostrou muito eficiente, inclusive permitindo uma posterior migração - compatível com a anterior - para uma implementação microkernel, com um nível muito maior de subdivisões de sistema na versão do OS/2 para a plataforma PowerPC. A versão microkernel, entratanto, nunca se tornou comercial, estando disponível apenas na forma de cópias "beta" das últimas versões produzidas nos laboratórios da IBM.

2.1.4. Modelo em Camadas do Windows NT / XP / Vista / 7

O Windows NT também é um dos sistemas que foi projetado corretamente seguindo a arquitetura em camadas (ao contrário de seus antecessores, Windows 95, 98 e Me). A arquitetura do Windows NT é muito similar ao das primeiras versões do OS/2 (que foi desenvolvido pela Microsoft para a IBM), embora tenha sido uma implementação microkernel desde o princípio (ao contrário do OS/2).

Entretanto, pode-se dizer que a Microsoft "errou na mão" ao projetar o Windows NT e criou um número de camadas tal que, com a eficiência do código gerado, tornava o sistema todo muito mais lento que o seu "irmão pobre", o Windows 9x (onde todos os acessos eram feitos quase que diretamente, similar ao modo de operação do MS-DOS).

Por esta razão, a Windows NT teve sua arquitetura um tanto modificada, tornando-o um híbrido entre sistema microkernel e monolítico e permitindo que uma camada passe por cima de outra em algumas operações para que o desempenho obtido seja adequado. Este processo de modificação iniciou-se no Windows NT 4.0 e foi concluído apenas no Windows 2000, a versão mais eficiente da tecnologia NT.

No Windows Vista e 7, nos quais a arquitetura foi relativamente remodelada, o sistema permanece com as características híbridas, mas caminha em direção a uma implementação microkernel pura.

3. CHAMADAS DE SISTEMA (*System Calls*)

* Um processo (programa) precisa gravar um arquivo...
  - Processo: modo usuário...
  - Rotina de arquivos: modo supervisor...
  - *System Call*!
    + Interrupção por Software (instrução da CPU)
* Toda CPU tem esse modos?

Uma chamada de sistema é, basicamente, o que um processo precisa fazer quando, por exemplo, ele deseja acessar um dispositivo ou mesmo gravar ou ler um arquivo. Neste momento, ele precisa pedir ao sistema operacional que realize a parte básica destas tarefas por ele. Mas, neste momento, uma pergunta deve estar no ar: se o núcleo e grande parte dos subsistemas são executados no *modo supervisor*, e um programa sendo executado no *modo usuário* só tem acesso à sua área de dados, como é possível ao programa do usuário acessar aos serviços dos subsistemas e do kernel?

A resposta é que, obviamente, existe um mecanismo para isso, muito embora não seja o mesmo para todos os sistemas e arquiteturas de computador. Todo processador que possui dois (ou mais) modos de operação permite uma maneira de um processo de um modo mais restrito (modo usuário) invocar função a serem executadas em um modo menos restrito (modo supervisor). Normalmente isso é feito através de alguma instrução em linguagem de máquina. No caso dos processadores 80x86, isso é realizado através da instrução de **interrupção por software**.

Obviamente, quando se pede para o sistema operacional executar uma função qualquer, é necessário passar parâmetros antes de invocá-lo (por exemplo, através da interrupção por software). Estes parâmetros podem ser passados por registradores, através do armazenamento destes dados na memória (indicando em um registrador qual endereço é esse) ou mesmo colocando os dados na pilha do processo corrente. O processo todo pode ser visualizado na figura 4 (MACHADO e MAIA, 2007).

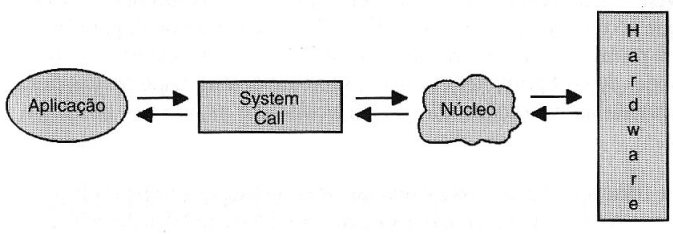


Figura 4: O processo de chamadas de sistema (System Calls)

4. MÁQUINAS VIRTUAIS (OPCIONAL)

* "Emulador"
    - Conversão de Solicitações de Acesso a Dispositivos
    - Java e .Net
* Virtualizador de Hardware
    - Divisão dos recursos do hardware implementada no hardware
    - Perda de desempenho desprezível

Uma máquina virtual nada mais é que um software que "emula" todos os níveis de um computador, permitindo que outros software sejam executados dentro dela (fornecendo todos os serviços necessários).

Quando o software em execução em uma máquina virtual faz alguma solicitação, a máquina virtual converte esta solicitação de alguma forma que o sistema hospedeiro (aquele que executa a máquina virtual) seja capaz de atender àquela requisição.

Neste sentido, é possível considerar o Sistema Operacional como uma máquina virtual, mas não apenas. A linguagem Java e a plataforma .Net possuem máquinas virtuais, já que interpretadores são, por natureza, máquinas virtuais. Emuladores dos mais diversos equipamentos são, também, máquinas virtuais.

Existem hoje também os "virtualizadores de hardware", nas plataformas de 64 bit, que permitem que vários sistemas operacionais de 32 bits sejam executados ao mesmo tempo, com uma perda de desempenho desprezível. Este modo é similar ao modo V86 (de Virtual 8086), que existe desde os processadores 80386 para a execução de aplicativos de 8086/8088 em paralelo.

5. BIBLIOGRAFIA

DAVIS, W.S. Sistemas Operacionais: uma visão sistêmica. São Paulo: Ed. Campus, 1991.

MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.

SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.

TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. S. Paulo: Prentice Hall, 2003.

Unidade 3: Processos e Gerência de CPU

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos básicos que definem um processo, suas características e o os critérios de escalonamento de tempo de CPU.

Bibliografia:

- MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.
- TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. São Paulo: Prentice Hall, 2003.
- SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.

INTRODUÇÃO

* Início: um computador = um programa
    - Acesso direto aos recursos
    - programas, tarefas, *jobs*...
* Problema: um computador = diversos programas?
    - Execução em ambiente "exclusivo"
    - *processo* : importante!
* Processo?
    - Programa: está no disco
    - "Cópia de Programa em Execução"
        + "código + estado"
* Mais processos que CPUs = troca de processo em execução
    - Quando processo finaliza
    - Quando processo espera algo ocorrer
    - Quando acaba o tempo do processo: *time slice* (fatia de tempo)
* Troca de processos: SO deve reconfigurar a CPU
    - Os dados de reconfiguração => o "tal" estado
        + Em que ponto da "receita" estávamos quando paramos?

No início, os computadores executavam apenas um programa de cada vez, que podiam acessar diretamente todos os seus recursos. Muitos nomes foram sugeridos para indicar as "coisas que o computador executa": programas, tarefas, *jobs*...

Entretanto, a partir do momento em que os computadores (e sistemas operacionais) passaram a executar diversos programas ao mesmo tempo, surgiu uma necessidade de criar uma "separação" entre estas tarefas, e dessa necessidade surgiu o conceito de *processo*, que se

tornou unânime. Neste contexto, o conceito de processo é o mais importante de um sistema operacional moderno.

Simplificadamente, "processo" é um programa em execução. Entretanto, quando se trabalha em sistemas multitarefa, essa definição fica simplista demais. Em um sistema operacional multitarefa e multiusuário, o sistema operacional simula, para cada processo sendo executado, que apenas ele está sendo executado no computador. Pode-se dizer que um "programa" é uma entidade passiva (como um arquivo no disco) e que cada cópia dele em execução é um processo, incluindo não apenas o código original (único), mas dados e informações de estado que podem ser únicas de cada processo.

Como, em geral, há mais processos sendo executados do que CPUs disponíveis no equipamento, isso significa uma troca constante e contínua do processo sendo executado por cada CPU. O tempo que cada processo fica em execução é denominado "fatia de tempo" (*time slice*). O tempo da CPU é compartilhado entre processos.

Ora, considerando que cada processo tem requisitos específicos, a cada troca de processo em execução é necessário reconfigurar uma série de propriedades do processador. Um processo, então, envolve não apenas o "programa em execução", mas também todos os dados necessários para que o equipamento seja configurado para que este processo incie (ou continue) sua execução. Assim, de uma forma mais abrangente, "processo" pode ser definido como o "um programa e todo o ambiente em que ele está sendo executado".

1. O MODELO DE PROCESSO

* Ambiente do Processo
    - Memória
        + Ganho 64K; Precisa de mais = ?
    - Ponto de Execução
    - Arquivos de Trabalho (Abertos)
    - Prioridades...
* Program Control Block (PCB) (Figura 1)
    - Estado do Hardware (Registradores, dispositivos etc.)
    - Estado do Software (Quotas, arquivos, "estado" do processo etc.)
    - Configuração da Memória (onde e quanto acessar)

Como foi dito, "processo é o ambiente onde se executa um programa", envolvendo seu código, seus dados e sua configuração. Dependendo de como o processo for definido, um programa pode funcionar ou não; por exemplo: se um programa for executado como um processo que permite que ele use apenas 64KB de RAM e este programa exigir mais, certamente ele será abortado.

Estas informações (sobre quanta RAM está disponível em um processo, em que ponto ele está em sua execução, qual seu estado e prioridades etc.) são armazenadas em uma

estrutura chamada "Bloco de Controle de Processos" (Process Control Block - PCB). A figura 1 apresenta um exemplo de estrutura de PCB:

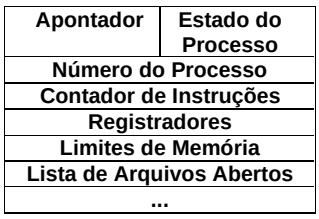


Figura 1: Exemplo de estrutura de um PCB

Os processos são gerenciados através de chamadas de sistema que criam e eliminam processos, suspendem e ativam processos, além de sincronização e outras ações.

A PCB usualmente armazena as seguintes informações: **estado do hardware**, **estado do software** e **configuração de memória**.

1.1. Estado do Hardware

As informações do estado do hardware são, fundamentalmente, relativas aos valores dos registradores dos processadores. Assim, o ponteiro da pilha (SP), do contador de programa (PC) e outros são armazenados quando um "programa sai de contexto", isto é, ele é momentaneamente interrompido para que outro possa ser executado.

Estas informações são armazenadas para que, quando o programa seja colocado novamente em execução, ele possa continuar exatamente onde estava, como se nada houvesse ocorrido e ele nunca tivesse deixado de ser executado.

Estas trocas, em que um processo deixa de ser executado, sua configuração é salva, e outro processo passa a ser executado (após devida configuração) são denominadas "trocas de contexto".

1.2. Contexto do Software

As informações de contexto do software são, essensialmente, identificações e definição de recursos utilizados.

Há sempre uma IDentificação de Processo (PID), que em geral é um número, podendo indicar também a identificação do criador deste processo, visando implementação de segurança.

Há também as limitações de quotas, como números máximos de arquivos que podem ser abertos, máximo de memória que o processo pode alocar, tamanho máximo do buffer de E/S, número máximo de processos e subprocessos que este processo pode criar etc.

Adicionalmente, algumas vezes há também informações sobre privilégios específicos, se este processo pode eliminar outros não criados por ele ou de outros usuários etc.

1.3. Configuração de Memória

Existe, finalmente, as informações sobre configuração de memória, indicando a região da memória em que o programa, onde estão seus dados etc. Mais detalhes sobre esta parte serão vistos quando for estudada a gerência de memória.

2. Estados de Processo

* Mais processos que CPU = fila de processos
* Quando um processo sai de execução, outro entra em execução
* Decisões dependem dos estados dos processos:
    - Novo
    - Execução (running)
    - Pronto (ready)
    - Espera/Bloqueado (wait / blocked)
        + Recurso próprio x de terceiros
    - Terminado
* Trocas
    - *Task Switch* (troca de processo)
        + Pronto => Execução
        + Execução => Pronto
    - Sincronia
        + Execução => Espera/Bloqueado
        + Espera/Bloqueado => Pronto
        - Sistemas Tempo Real: Espera/Bloqueado = > Execução!

Em geral, em um sistema multitarefa, há mais processos sendo executados do que CPUs. Desta forma, nem todos os processos estarão sendo executados ao mesmo tempo, uma boa parte deles estará esperando em uma *fila de processos*.

Em algum momento, o sistema irá paralisar o processo que está em execução (que irá para fila) para que outro processo possa ser executado ("sair" da fila). Os processos podem, então, estar em vários estados:

- Novo
- Execução (running)
- Pronto (ready)
- Espera/Bloqueado (wait / blocked)
- Terminado

O processo *novo* é aquele que acabou de ser criado na fila e ainda está em configuração. Funciona, para o resto do sistema, como um processo em *espera* - ou seja, ainda não pode ser executado.

O processo em *execução* é aquele que está ativo no momento (em sistemas com mais de uma CPU, mais de um pode estar neste estado ao mesmo tempo). Este processo é considerado "fora da fila", apesar de na prática ele ainda estar lá.

O processo *pronto* é aquele que está apenas aguardando na fila, esperando por sua "fatia de tempo" (*timeslice*) para ser executado.

O processo está em *espera* quando está na fila por aguardar algum evento externo (normalmente a liberaçao ou resposta de algum outro processo ou de algum dispositivo). Quando um processo espera pela liberação de um recurso, em geral se diz que ele está *bloqueado*. Quando o processo espera uma reposta de um recurso ao qual ele já tem acesso, diz-se que ele está simplesmente em *espera*.

O processo *terminado* é aquele que acabou de ser executado, mas está na fila porque ainda está sendo eliminado. Funciona, para o resto do sistema, como um processo em *espera* - ou seja, não pode ser executado.

Processos em espera ou prontos podem ser colocados na memória secundária (swap), mas para estar em execução ele sempre precisará ser movido para a memória principal.

2.1. Mudanças de Estado

O estado de um processo é alterado constantemente pelo sistema operacional; na simples troca de contexto, ou seja, qual processo está sendo executado, ocorre essa alteração de estado. Estas trocas podem ser:

Task Switch (troca de contexto)
- Pronto => Execução
- Execução => Pronto

Sincronia
- Excução => Espera/Bloqueado
- Espera/Bloqueado => Pronto

Apenas em sistemas Tempo de Real (*Real Time*) é permitido que um processo em espera ou bloqueado volte diretamente para o estado de execução.

3. COMUNICAÇÃO ENTRE PROCESSOS (OPCIONAL)

* E quando um processo precisa de informações de outro?
    - Rede, por exemplo!
    - Sistema operacional permite troca de informações
    - Cuidado: *deadlocks*!
* Será visto com mais detalhe no fim do curso

Muitas vezes já existe um processo sendo executado que realiza uma determinada tarefa de interesse como, por exemplo, se comunicar com a rede. Assim, se um programador desenvolve um software que precisa de dados advindos da rede (um navegador web, por exemplo) veria com bastante interesse a possibilidade de se comunicar com o processo que já faz o acesso à rede para receber dados da mesma.

Isto é interessante, possível e viável, mas pode gerar alguns problemas, como o já citado *impasse* ("*deadlock*"), em que um primeiro processo espera por uma resposta de um segundo processo que, por sua vez, para poder responder, espera uma resposta do primeiro processo.

Em uma outra situação, suponha que um processo esteja trocando informações com outro processo através de um buffer (um *pipe*, por exemplo). O processo que irá ler os dados precisa esperar que haja dados no buffer para que ele seja lido, ou até mesmo precisa esperar que o buffer esteja cheio (quando então o processo que escreve no buffer precisa esperar que ele seja esvaziado). O sistema operacional atua, então, bloqueando e reativando os processos adequadamente, nos momentos corretos.

Quando a comunicação com o espaço de dados é feita diretamente, sem intervenção do sistema operacional (como no caso do uso de uma região de memória compartilhada - *shared memory*), é necessário o uso, por exemplo, de semáforos, que é um recurso do sistema operacional que pemite que um processo diga que está usando um recurso; sempre que ele for precisar do recurso, ele pede o uso deste recurso e, se o recurso já estiver sendo usado, o sistema operacional o bloqueará até que o recurso esteja disponível.

4. PROCESSOS x THREADS (OPCIONAL)

* Subprocesso: um processo criado por outro processo
    - Processo filho: é fechado quando o processo pai é fechado
    - Todo processo é "filho" do processo principal do SO
* Thread?
    - Não tem características de Ambiente próprias
    - Mais leve

Nos tempos atuais, com o advento da multitarefa pervasiva, muito se houve falar em processos, subprocessos e threads. Mas o que significam estas palavras?

Resumidamente, um subprocesso é um processo normal, mas cuja criação foi solicitada por um outro processo já existente. Em geral, com excessão da shell (que normalmente é o processo 0), todos os outros processos sendo executados são, na realidade, subprocessos. Isso ocorre porque os processos do usuário são inciados, normalmente, por intermédio da shell do sistema operacional. Uma razão para um processo qualquer criar um outro subprocesso é a necessidade (ou a possibilidade) de executar partes distintas do programa em paralelo, uma delas colaborando com a outra para a execução de um trabalho mais complexo.

Normalmente, os processos "filhos" compartilham algumas informações com seus processos "pai": todos os arquivos abertos no processo pai permanecem abertos no proesso filho, por exemplo. Isso facilita sensivelmente a cooperação entre eles. Além disso, quando um processo "pai" é finalizado, seus processos filhos também o serão. Assim, se um processo criou vários subprocessos e ele for finalizado, seus subprocessos também o serão.

Mas onde os threads se encaixam nisso? Bem, com o tempo se percebeu que a criação de subprocessos para executar tarefas de um mesmo programa trazia benefícios, mas consumia recursos desnecessários demais, já que uma nova região de memória precisava ser alocada, além da criação de uma nova entrada na tabela de processos, com o armazenamento de todas as informações de estado de hardware, software e configuração de memória.

Para evitar este consumo excessivo (e desnecessário) de recursos, criou-se o conceito de thread: uma nova linha de execução para um mesmo processo. Mas o que isso significa? Significa que um mesmo processo pode ter vários estados de hardware, todos eles compartilhando os mesmos recursos do contexto de software e configuração de memória. Isso economiza espaço na PCB **e** evita que um novo bloco de memória precise ser alocado.

Assim, implementações de threads bem feitas tornam aplicativos que implementam paralelismo substancialmente mais "leves" do que aplicativos que implementam o mesmo paralelismo através de subprocessos.

5. ESCALONAMENTO DE PROCESSOS (*Process Scheduling*)

* Gerenciador de Processos
    - Cria e Remove processos
    - Escalonador => troca processos
        + Usa informação do processo para isso
            = estado: pronto
            = prioridade
        + Evitar *starvation*

Uma das principais funções do Gerenciador de Processos de um sistema operacional multitarefa/multiusuário é realizar o Escalonamento de Processos, ou seja, realizar a troca de contexto dos processos. Esta tarefa pode ser executada de muitas maneiras, com auxílio ou não dos processos.

Basicamente, existe um número limitado de CPUs e um número usualmente maior de processos prontos para execução. O **escalonador** (*scheduler*) é o elemento do gerenciador de processos responsável pelo *escalonamento*, e sua função é **escolher** qual dos processos em estado de "pronto" será o próximo a receber o estado de "execução" na próxima fatia de tempo de execução (*timeslice*).

Seus objetivos são manter a CPU ocupada a maior parte do tempo (enquanto houver processos a serem executados, claro), além de balancear seu uso, ou seja, dividir o tempo entre os processos de acordo com sua *prioridade*, além de garantir um tempo de resposta razoável para o usuário e evitar que algum processo fique tempo demais sem ser executado (*starvation*).

Como um guia, o escalonador utiliza informações sobre os processos (se ele usa muito I/O, se ele usa muita CPU, se ele é interativo, se não é interativo etc.) para tentar acomodar da melhor maneira possível as necessidades.

5.1. Critérios de Escalonamento

* Critérios Comuns para Nortear Escalonamento
  - Uso da CPU : pagamento tradicional (todas as tarefas prontas)
  - Execuções Completas (*Throughput*) : pagamento fixo por tarefa
  - Tempo total de execução (*Turnaround*) : pagamento por tempo de cpu
  - Tempo de resposta : o que importa é a sensação do usuário
* Dois tipos de escalonamento
  - Não preemptivos
  - Preemptivo

Os escalonadores costumam avaliar diversos critérios para definir qual dos processos colocar em execução a cada troca de contexto. Alguns destes critérios são:

- Utilização da CPU: em geral se deseja maximizar
- Throughput: tarefas executadas em um intervalo de tempo. Desej-se maximizar.
- Turnaround: tempo total de execução do processo. Deseja-se minimizar.
- Tempo de Reposta: tempo de uma requisição do usuário até a resposnta. Deseja-se minimizar.

Além disso, existem alguns tipos básicos de escalonamento, que serão vistos a seguir: escalonamento não-preemptivo, escalonamento preemptivo e escalonamento com múltiplos processadores.

5.2. Escalonamento Não-Preemptivo

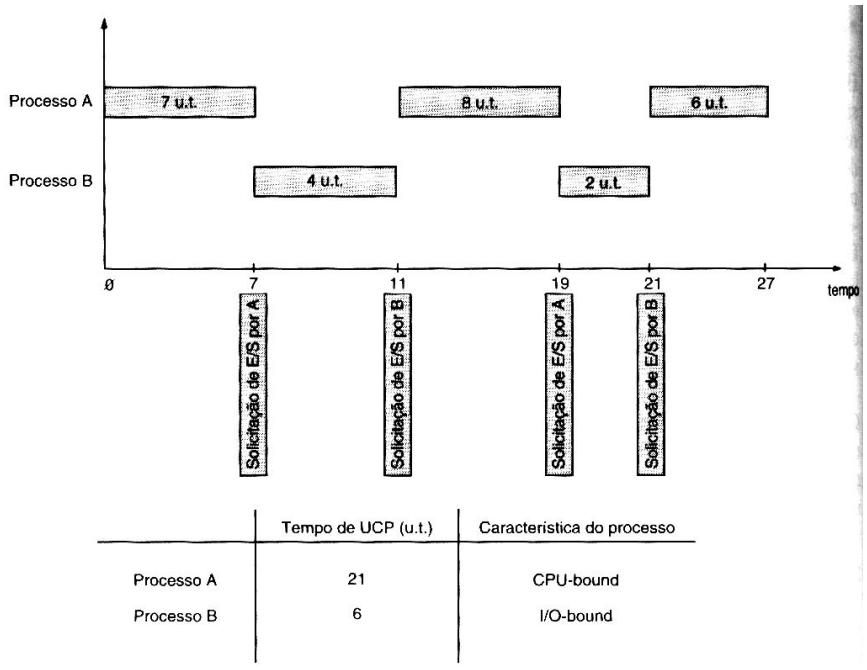
* Mais antigo: sistema não interrompe aplicativo
* Quando pode ocorrer troca
  - processo finalizado
  - processo entra sozinho em espera ou bloqueio (esperando algo)
* Tipos de Algoritmos
  - Fila FIFO x Shortest Job First x Cooperativo (Fila de Mensagens)
* Vantagens (simplicidade) x Desvantagens (uso desigual de CPU, congelamento)

O escalonamento não-preemptivo é o tipo mais antigo, criado no momento em que mais de um programa podia ser carregado na memória (como nos primeiros sistemas batch). Neste tipo de escalonamento, quando um processo adquire o direito de usar a CPU, nenhum outro processo pode lhe tirar este recurso.

A troca de contexto ocorre em apenas duas situações básicas: a) quando o processo em execução é finalizado ou b) quando o processo em execução entra em estado de espera ou bloqueio (aguardando uma operação de I/O, por exemplo).

Alguns algoritmos possíveis para o escalonamento não-preemptivo são:

- **Escalonamento FIFO (First In, First Out)**: neste esquema, o primeiro processo a ser carregado é o que será executado primeiro. O problema desta abordagem é que normalmente o tempo de processamento fica mal distribuído, como pode ser visto na figura 2 (MACHADO e MAIA, 2007).



	Tempo de UCP (u.t.)	Característica do processo
Processo A	21	CPU-bound
Processo B	6	I/O-bound

Figura 2: Escalonamento FIFO

- Escalonamento SJF (Shortest Job First): neste esquema, sempre que for possível ocorrer uma troca de contexto, o sistema selecionará para colocar em execução o processo irá ser finalizado mais rapidamente, dentre aqueles com estado de "pronto". O problema desta abordagem é justamente determinar o tempo de execução de um processo.

- Escalonamento Cooperativo: neste esquema, não há qualquer intervenção do sistema operacional para interromper um processo ou trocar o contexto, a não ser que o próprio aplicativo libere o processamento. Nestes sistemas existe, em geral, algum mecanismo para que o processo permita que o sistema operacional realize a troca de contexto. No caso dos Windows 3.x ou MacOS anteriores à versão 8, os programas precisam periodicamente consultar uma fila de mensagens (para verificar se o usuário entrou com algum comando, por exemplo), fila esta administrada pelo sistema operacional, que usa, então, este momento para permitir a troca de contexto, ou o task switch, como também é chamado. O problema desta abordagem é que, se por alguma razão um programa não consultar esta fila de mensagens (por exemplo, entrar em um loop infinito), não será realizada qualquer troca de contexto e o sistema poderá congelar.

5.3. Escalonamento Preemptivo

* Mais comum na atualizada: sistema é dono do computador
* Quando pode ocorrer troca
  - mesmas situações do não-preemptivo
  - fim do *time slice*
  - algum evento externo (*real time*)
* Tipos de Algoritmos
  - Round-Robin, Por Prioridades, Múltiplas Filas, Tempo Real
* Vantagens (melhor distribuição de CPU) x Desvantagens (*overhead*)

O escalonamento preemptivo é o tipo mais comum atualmente. Neste tipo de escalonamento, é o sistema operacional quem "decide" o momento de interromper um processo para que outro possa ser executado. Isto pode ocorrer em momento similares aos dos sistemas não-preemptivos, como o momento de uma operação de I/O ou na consulta da fila de mensagens, mas também pode ocorrer assim que um tempo pré-determinado tenha se passado, determinando o que se chama de *timeslice*, a fatia de tempo daquele processo.

Este tipo de escalonamento permite que processos com maior prioridade sejam atendidos sempre que necessário, e os tempos de resposta do sistema às requisições do usuário também se tornam melhores. Adicionalmente, como existe um tempo máximo de processamento para os processos comuns, em geral o tempo do processador é melhor distribuído entre os diversos processos.

Entretanto, é preciso ressaltar que o processo de preempção tem um custo computacional, isto é, causa um *overhead*. Assim, os critérios de preempção precisam ser muito bem definidos, sob pena de desperdiçar recursos com trocas desnecessárias de tarefas.

Alguns algoritmos possíveis para o escalonamento preemptivo são (lembrando que muitos sistemas utilizam vários destes esquemas de maneira combinada):

- Escalonamento Circular (round robin): este sistema é a versão "preemptiva" do sistema FIFO, em que cada processo terá parte de sua execução na ordem em que os processos foram carregados. Assim, o primeiro processo carregado será marcado como em "execução". Quando ele liberar o processamento (esperando uma operação de I/O, por exemplo) **ou** seu **timeslice** acabar, o sistema operacional o colocará no estado "pronto" e ativará o processo seguinte na fila, indicando-lhe o estado de em "execução". Isso ocorrerá continuamente até chegar ao último processo da fila, quando então a execução volta para o primeiro processo. Nesta abordagem a distribuição de tempo é um pouco melhor que nos esquemas não preemptivos, como pode ser visto na figura 3 (MACHADO e MAIA, 2007).

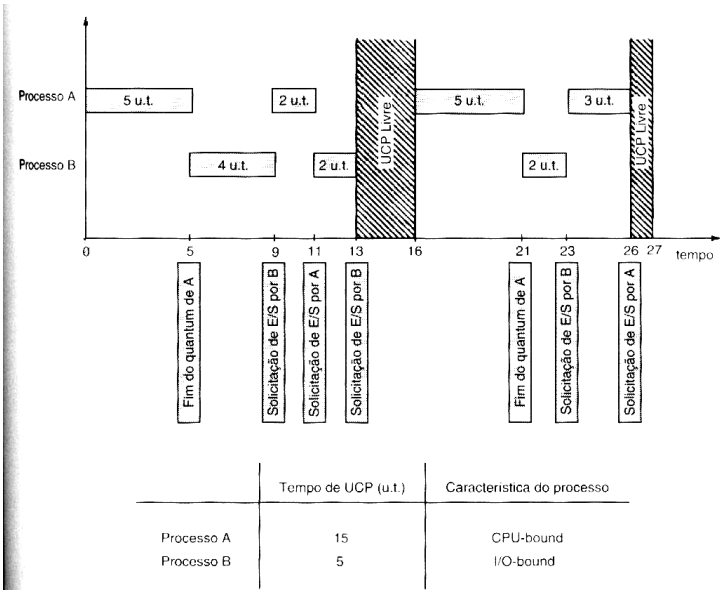


Figura 3: Escalonamento Circular

- Escalonamento por Prioridades: apesar do escalonamento circular distribuir melhor o uso do processador, ele trata todos os processos como iguais. Isto quer dizer que não é possível definir um processo com mais prioridade que outros no esquema circular. Para permitir o sistema de prioridades, foi criado o escalonamento por prioridades. O escalonamento por prioridades funciona da seguinte forma: sempre que se tornar "pronto" um processo com maior prioridade do que o em "execução", o processo em execução será interrompido e aquele com maior prioridade será executado. Normalmente este sistema é implementado em conjunto com o circular e, a cada preempção, o sistema reavalia a ordem de execução dos processos (não se atendo à seqüência FIFO).

Existem dois tipos de escalonamento de prioridade: o de **prioridade dinâmica e o de prioridade estática**. Na prioridade estática um processo recebe uma prioridade e a mantém por toda sua execução. É um esquema simples, porém pode gerar tempos de resposta altos. Quando se usa prioridade dinâmica, por outro lado, o sistema operacional ajusta a prioridade do processo de acordo com o funcionamento do sistema. Por exemplo:

- cada programa em estado "pronto" recebe um incremento de prioridade a cada preempção em que não é escolhido para executar;
- o processo que acabou de sair do estado de execução tem sua prioridade reduzida;
- um processo que estava em espera e se tornou pronto tem um aumento significativo em sua prioridade (para compensar o tempo que ficou parado esperando).

- Escalonamento por Múltiplas Filas: é uma forma de combinar o escalonamento por prioridade com o escalonamento circular. Há diferentes filas de processamento, cada uma delas para processos de uma prioridade específica. É possível ter, por exemplo, uma fila para prioridade "tempo real", uma fila para prioridade "regular" e uma fila para prioridade "ociosa". Processos de uma fila de menor prioridade só serão colocados em execução se todos os processos das filas de maior prioridade estiverem em modo de espera.

- Escalonamento de Sistemas de Tempo Real: é uma forma específica do escalonamento por prioridades onde não há *timeslice* algum. O processo sendo executado é sempre o de maior prioridade e as reavaliações de processos (para troca de contexto) são disparadas por dois tipos de eventos: a) processo em execução (de maior prioridade) foi finalizado ou entrou em espera; e b) um sinal externo (interrupção) foi disparada por algum dispositivo (que pode ter feito com que um processo de mais alta prioridade saia do estado de bloqueio/espera).

5.4. Escalonamento com Múltiplos Processadores

* Sistemas Fracamente Acoplados (Sistema Distribuído)
    - Cada equipamento tem seu próprio escalonador
* Sistema Fortemente Acoplado (Sistema Paralelo)
    - Escalonador tem mais trabalho
    - Escalonador em CPU exclusiva
        + Mestres x Escravos
    - Escolher CPU para um processo
        + Escolha fixa x dinâmica
        + CPUs podem ser diferentes?
    - Não alocar mais de uma CPU para threads do mesmo processo!

Quando se tem um sistema distribuído (chamados de "fracamente acoplados"), cada equipamento tem seu próprio escalonador e funcionam quase que como equipamentos independentes.

Quando se tem um sistema paralelo local (fortemente acoplado), o escalonador é responsável por designar para qual processador um processo sera alocado em cada *timeslice*.

Alguns sistemas mantém fixamente um processo a um dado processador; outros, entretanto, alocam um processo para qualquer um deles, para o primeiro que ficar disponível.

O único cuidado que precisa ser tomado é o de não alocar o mesmo processo para mais de um processador ao mesmo tempo, caso mais de um esteja disponível no momento da troca de contexto.

6. BIBLIOGRAFIA

DAVIS, W.S. Sistemas Operacionais: uma visão sistêmica. São Paulo: Ed. Campus, 1991.

MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.

SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.

TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. S. Paulo: Prentice Hall, 2003.

Unidade 4: Gerência de Memória

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos básicos do gerenciamento de memória realizado pelo sistema operacional.

Bibliografia:

- DAVIS, W.S. Sistemas Operacionais: uma visão sistêmica. São Paulo: Ed. Campus, 1991.
- MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.
- TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. São Paulo: Prentice Hall, 2003.
- SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.

INTRODUÇÃO

* Memória RAM => Recurso limitado
  - Tamanho e Velocidade
  - Seu gerenciamento é **muito** importante
* Início: apenas um processo: gerenciamento simples!
* Multitarefa/Multiusuário: cada byte conta!

Memória principal (RAM) sempre foi um dos recursos mais limitados em um sistema computacional, seja em termos de velocidade ou em termos de quantidade. Por esta razão, um eficiente sistema de gerenciamento de memória sempre foi uma das grandes preocupações no projeto de um sistema operacional.

O gerenciamento de memória principal era, inicialmente, simples. Os computadores executavam apenas um processo de cada vez e não eram necessários processos mais complicados para gerenciarmento de memória. Entretanto, à medida em que os sistemas multitarefa e multi-usuários surgiram, a gerência de memória se tornou mais complexa, com o objetivo de aproveitar melhor cada byte disponível.

Assim, serão apresentados os fundamentos da gerência de memória, desde os processos mais simples aos mais atuais, além de apresentar o conceito de endereçamento virtual e memória virtual.

1. ALOCAÇÃO CONTÍGUA SIMPLES

* Sistemas monoprogramados
  - Existe isso?
* Dois processos: Programa + SO
  - Figura 1
* Processo: Respeitar o limite de memória
* Primeiras CPUs: nenhum controle de acesso
* CPUs posteriores: MMU: Memory Management Unit
  - Registrador que indica fim da área de sistema (Figura 2)
    + Programa do usuário não tem acesso lá!
      = *Access Violation* ou GPF
    + Fixo x Modificável no Modo Kernel
  - Modo Kernel: abaixo do valor do registrador da MMU
    + Inicialmente MMU aponta para topo da memória (tudo Kernel)
* Problema: ineficiente e inviável para multitarefa

Em sistemas monoprogramados, ou seja, aqueles em que há apenas um processo executando (e é necessário que ele termine para que outro seja carregado), não há a necessidade de um gerenciamento complexo.

Neste tipo de sistema, em geral, o sistema operacional é carregado em uma dada região da RAM (normalmente a parte mais baixa) e o programa sendo executado (o processo) é carregado em seguida ao sistema operacional, sempre na mesma área (Figura 1). O processo so precisa se preocupar em não ultrapassar o limite de memória disponível.

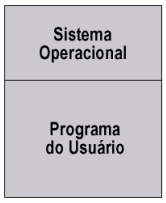


Figura 1: Alocação Contígua Simples

Nos primeiros equipamentos não existia qualquer proteção de memória para a área do sistema operacional. Isto significa que, se o processo executado sobrescrevesse a área do sistema operacional, poderia danificá-lo.

Em equipamentos posteriores, os processadores foram dotados de uma unidade simples, chamada MMU (*Memory Management Unit* - Unidade de Gerenciamento de Memória), que tinha como responsabilidade limitar o acesso do software à região do sistema oepracional, criando uma primeira versão simples de "modo supervisor" e "modo usuário".

Nestes primeiros sistemas o único controle existente na MMU era um registrador que indicava o endereço de memória onde acabava a área de sistema, ponto em que se iniciáva a área do software do usuário (Figura 2).

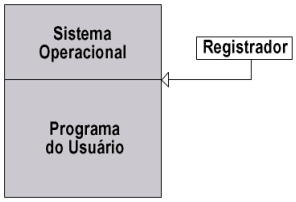


Figura 2: Separação entre Área de Sistema e Área de Usuário nas primeiras MMU

Em essência, qualquer código que fosse executado em uma área abaixo da área de sistema (endereço de memória menor ou igual ao valor contido no registrador da MMU) teria acesso à toda a memória; por outro lado, qualquer programa que fosse executado em um endereço **acima** do valor contido no registrador da MMU só poderia acessar a área acima da MMU; isto é, se o programa na área do usuário tentar escrever na área do sistema, um erro do sistema seria gerado, em geral causando o fechamento do aplicativo com uma mensagem do tipo *Access Violation* (Violação de Acesso) ou *General Protection Fault* (Falha de Proteção Geral).

Apesar de bastante simples e eficaz, este método de gerenciamento é ineficiente, muitas vezes sobrando uma grande parcela de memória inutilizada (nem pelo sistema, nem pelo software do usuário). Além disso, este esquema de gerenciamento é inviável em sistemas multitarefa ou multi-usuários.

2. ALOCAÇÃO PARTICIONADA

* Sistemas multiprogramados
  - Sistemas autais
* MMU tem vários registradores de acesso à memória
* Memória Dividida em Regiões
  - Cada processo fica com uma!
* Particionamento Estático x Dinâmico

Como objetivo de permitir que vários programas pudessem ser executados simultaneamente, foi criado o esquema de alocação particionada. Neste esquema a memória é dividida em regiões, cada região estando disponível para um processo.

Há duas formas de particionar a memória: estaticamente o dinamicamente.

2.1. Particionamento Estático

* Definição das regiões no momento do *boot*
  - Ex.: Figura 3
  - E para alocar um programa de 10KB?
    + Reiniciar sistema
* Memória Absoluta
  - Programas feitos para rodar em uma região só podem rodar nela
  - Solução: Programas Relocáveis
    + "Loader" ajusta o programa para ser executado em sua região
* Armazenamento do Mapa da Memória (Figura 3a)
  - MMU: Registradores = número de partições possíveis
* Vantagem: simplicidade
* Problema: ineficiência: alguns poucos processos minúsculos esgotam a memória

Uma das primeiras maneiras de particionar a memória foi o particionamento estático. Neste tipo de particionamento, as divisões da memória (e seus tamanhos) eram determinadas no momento do *boot*, isto é, quando o sistema é ligado. Se alguma alteração fosse necessária neste particionamento, uma reinicialização do sistema seria necessária.

Assim, se fosse definido que a memória seria dividida em 4 regiões: 5KB para o sistema operacional, 2KB como a partição A, 5KB como a partição B e 8KB como a partição C (figura 3), um programa de 10KB nunca poderia ser executado, pois ele não cabe em nenhuma das partições existentes. Para acomodá-lo, seria necessário modificar o particionamento de memória e reiniciar.

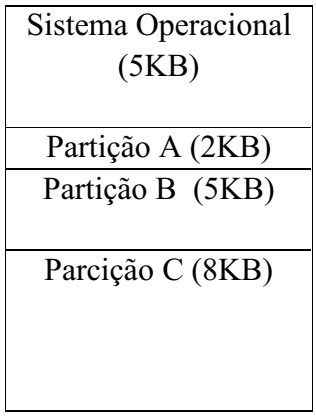


Figura 3: Particionamento Estático da RAM

Mas este não era o único problema deste mecanismo de particionamento estático. Inicialmente, o particionamento estático era também absoluto; isto é, os programas podiam ser executados apenas em uma dada região da memória. Assim, além do programa precisar ser menor que uma partição, esta partição deveria estar posicionada no local específico para o qual o programa foi compilado.

Este problema foi resolvido melhorando os compiladores e o "*loader*", o carregador de programas do sistema operacional, que passaram a possibilitar programas *relocáveis*, isto é, programas que, ao serem carregados, podem ter seus endereços corrigidos para que possam ser executados em qualquer região da memória RAM. Isso facilitou bastante a operação, porque agora bastava encontrar uma partição em que o programa coubesse para que ele pudesse ser executado.

De qualquer forma, neste sistema, o sistema operacional passou a ter de armazenar, em uma tabela, quais partições estavam disponíveis e quais não, numa tabela similar a esta:

Partição	Tamanho	Livre
1	2KB	Sim
2	5KB	Não
3	8KB	Não

Figura 3a: Mapa da Memória.

Assim, quando um programa é carregado em um sistema deste tipo, esta tabela é percorrida para encontrar qual a região disponível que melhor se adapta ao processo, para que ele pudesse ser carregado.

Neste sistema, a MMU passou a ter um conjunto de registradores igual ao número máximo de partições possíveis, sendo em cada um deles indicado o **tamanho** da partição, como indicado na Figura 3a.

Entretanto, apesar de este sistema permitir um melhor aproveitamento da memória com o carregamento de vários programas, fica claro que ele não é um "exemplo" em eficiência de alocação de memória: com a divisão de partições estabelecida acima, se 3 processos de 1 byte fossem carregados, a memória estaria totalmente "ocupada" de acordo com o gerenciamento do sistema operacional, ainda que fisicamente houvesse praticamente 15KB livres. Além disso, a memória é naturalmente fragmentada neste sistema, já que não é possível carregar um processo maior que uma partição, ainda que ele pudesse ser encaixado em duas ou mais partições contíguas livres.

Por estas razões, este modelo evoluiu para o modelo de alocação particionada dinâmica.

2.2. Particionamento Dinâmico

* MMU com dois registradores: **início** e **fim**
* Área tem exatamente o tamanho necessário
  - Ex.: Figura 4
  - Ou programa "cabe" ou "não cabe" na RAM
* Novo problema: fragmentação (Figuras 5a e 5b)
  - Há 6KB disponível
  - Falta memória para processo de 6KB
  - Memória **contígua**
* Solução: desfragmentar memória (Figura 6)
  - Mas memória continua absoluta
  - Relocação em tempo de execução: lento!
  - Estratégia para reduzir a necessidade de desfragmentação

Neste modelo foram eliminados os registradores de tamanho de partições de memória da MMU, sendo substituído por um sistema de **dois registradores**, um que marca o **início** e outro que marca o **fim** da área do processo, sendo seus valores determinados no momento de carregamento de um processo, exetamente do tamanho necessário para aquele programa (Figura 4). É claro que se um processo tentar acessar qualquer região fora de sua área permitida, um erro de violação de acesso será gerado.

Esta "pequena" mudança prontamente resolveu os dois problemas anteriores: o aproveitamento da memória poderia ser total e passou a não existir mais o problema de um processo não se encaixar em uma partição; com esta forma de gerenciamento existem apenas duas opções: ou o programa cabe na RAM ou ele não cabe na RAM.

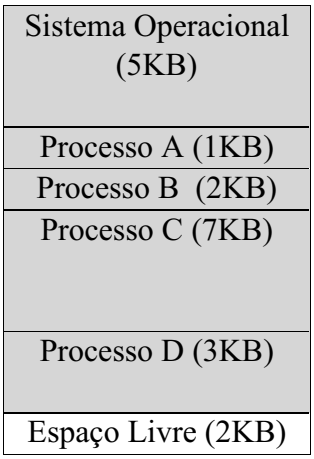


Figura 4: Particionamento Dinâmico da RAM

Entretanto, o problema da fragmentação não é resolvido diretamente por este mecanismo de alocação e ele surge no momento em que os programas começam a ser finalizados. Suponha que a memória ficou completamente cheia de pequenos programas de

1KB (Figura 5a) e, posteriormente, alguns foram fechados, liberando pequenos trechos de 1KB espalhados pela RAM, espaços estes que, somados, totalizam 6KB (Figura 5b).

Sistema Operacional
Processo A (1KB)
Processo B (1KB)
Processo C (1KB)
Processo D (1KB)
Processo E (1KB)
Processo F (1KB)
Processo G (1KB)
Processo H (1KB)
Processo I (1KB)
Processo J (1KB)

Sistema Operacional
Processo A (1KB)
Livre (2KB)
Processo D (1KB)
Livre (2KB)
Processo G (1KB)
Livre (2KB)
Processo J (1KB)

Figuras 5a e 5b: Fragmentação da RAM no particionamento dinâmico.

Neste caso, se o sistema precisar carregar um processo de 6KB, não será possível, já que os processos precisam de memória **contígua**, já que a proteção de memória é feita apenas com 2 registradores, um indicando o início e outro o fim da região protegida.

Assim, para corrigir este problema, foi criado o sistema de gerenciamento de particionamento dinâmico com relocação, em que os processos são relocados quando necessário (Figura 6), criando espaço para o novo processo.

Entretanto, este processo de relocação é bastante custoso computacionalmente (mesmo nos tempos atuais). Para evitar isso, os sistemas usam estratégias para minimizar a fragmentação da memória.

Sistema Operacional
Processo A (1KB)
Processo D (1KB)
Processo G (1KB)
Processo J (1KB)
Livre (6KB)

Figura 6: Desfragmentação da RAM no particionamento dinâmico com relocação.

2.3. Estratégias de Alocação Partição em Memória

* Diferentes Estratégias de Alocação
* Características dos Espaços Vazios
* Best-Fit: aloca no espaço mais próximo do tamanho necessário
  - Muitas áreas pequenas: pode aumentar a fragmentação geral
  - Busca lenta
* Worst-Fit: aloca no maior espaço disponível
  - Aumenta número de grandes áreas: diminui fragmentação
  - Busca lenta
* First-Fit: aloca no primeiro espaço disponível
  - Fragmenta o início da memória, mas deixa regiões grandes no final
  - Busca rápida

Em qualquer sistema de partição de memória, o gerenciador de memória manterá uma tabela onde ele será capaz de identificar as regiões livres da memória e seus tamanhos. Através destes dados, o gerenciador pode adotar uma de três estratégias mais comuns para alocação de memória: *best-fit*, *worst-fit* e *first-fit*.

2.3.1. Estratégia Best-Fit

Este mecanismo busca encontrar o espaço livre cujo tamanho é o mais próximo possível do espaço sendo alocado, minimizando o tamanho do espaço sem utilização naquele bloco.

Apesar de aparentemente bom, este sistema acaba aumentando o número de pequenas áreas de memória disponível espalhadas pela RAM, **aumentando** a fragmentação.

2.3.2. Estratégia Worst-Fit

Este mecanismo busca encontrar o espaço livre cujo tamanho é o mais díspar possível do espaço sendo alocado, maximizando o tamanho do espaço sem utilização naquele bloco.

Apesar de aparentemente ruim, este sistema acaba aumentando o número de grandes áreas de memória disponível na RAM, **diminuindo** a fragmentação.

2.3.3. Estratégia First-Fit

Neste sistema, a primeira região livre cujo espaço for suficiente será usada para a alocação do espaço. Este sistema, além de ser o menos custoso computacionalmente, em geral consegue bons resultados pois, como ele sempre tenta alocar o espaço no início da memória, quase sempre existem grandes áreas disponíveis no topo (fim) da memória.

3. ARQUIVO DE TROCA (*Swapping*)

* Processos precisam sempre estar na memória?
    - Quando estão executando a todo instante?
    - Quando estão esperando algo que pode demorar?
* Memória não é ilimitada!
    - Processo em longa espera: tirar da memória, temporariamente
    - Swapping: usar disco como memória-depósito-temporária
    - Volta à memória: quando processo for executar novamente
        + Será no mesmo lugar?
        + Relocação em tempo de execução: lento!

O sistema de alocação dinâmica apresentado anteriormente é muito interessante para aproveitar bem a memória. Entretanto, se ele for usado da forma descrita acima (o que, de fato, ocorre em muitos sistemas) ele tem uma severa limitação: todos os processos precisam permanecer na memória principal desde seu carregamento até o momento em que é finalizado.

Isso significa que os processos ocuparão a memória principal ainda que não estejam sendo processados como, por exemplo, quando estão esperando a resposta de algum dispositivo. Eventualmente, um processo pode esperar por horas para receber um dado sinal e, durante todo este tempo sem executar, estará ocupando a memória principal.

Isto não é exatamente um problema caso um sistema disponha de memória princial praticamente ilimitada, onde nunca ocorrerá um problema de falta de memória; entretanto, na prática, em geral a memória principal é bastante limitada e pode ocorrer de um novo processo precisar ser criado mas não existir espaço na memória. Neste caso, se houver processos em modo de espera, seria interessante poder "retirá-los" da memória temporariamente, para liberar espaço para que o novo processo possa ser executado.

Esta é exatamente a função do mecanismo de "swapping": quando um novo processo precisa ser carregado e não há espaço na memória principal, o processo menos ativo (em geral, bloqueado ou em espera) será transferido para um arquivo em disco, denominado **arquivo de troca** ou **arquivo de swap**, no qual ele permanecerá até o instante em que precisar ser executado novamente. Com isso, uma área na memória principal é liberada para que o novo processo seja executado.

Entretanto, o uso da técnica de *swapping* cria um problema: quando um processo é colocado no swap, ele não pode ser executado. Assim, ele precisará **voltar à memória** principal para poder ser executado, quando seu processamento for liberado. Isso não traz nenhum problema a não ser o fato de que dificilmente ele poderá voltar a ser executado na mesma região da memória principal em que estava inicialmente. Isso implica que ele deve ser **realocado dinamicamente**, da mesma forma com que contece nas desfragmentação da

memória, já citada anteriormente. E, também neste caso, este é um processo complexo e lento se tiver de ser executado por software (além de depender que o programa tenha sido preparado para isso).

4. ENDEREÇAMENTO VIRTUAL

* Como evitar ter de mexer em todos os endereços de um programa?
  - Fazer com que ele pense que está sempre executando no mesmo lugar!
* MMU: **registro de endereço base**
  - Programa pensa sempre que está rodando a partir do endereço zero
  - Processador calcula endereço real:
    endereço indicado pelo programa **+** endereço base
  - Exemplos: tabela

Como vários processos de gerenciamento de memória esbarram no problema de relocação dinâmica, com necessidades de ajustes lentos em todos os programas em execução, é natural que os desenvolvedores de hardware tenham se dedicado a criar mecanismos que eliminassem esses problemas.

Com o objetivo específico de facilitar este processo de realocação dinâmica para o swapping e, em parte, na desfragmentação de memória, as MMUs dos processadores passaram a incorporar, além dos registradores de início e fim da área de operação do processo, o **registrador de endereço base**. Este registrador indica qual é a posição da memória inicial de um processo e seu valor é somado a todos os endereços referenciados dentro do processo.

Assim, não só os processos passam a enxergar apenas a sua própria região da memória, como todos eles acham que a sua região começa sempre no endereço ZERO (0). Como consequência, os endereços do software não precisam ser mais modificados, pois eles são sempre **relativos** ao "zero" da memória que foi destinada a ele.

Em outras palavras, se um programa é carregado no endereço 1000 da memória, o registrador de endereço base é carregado com o valor 1000. A partir de então, quando o processo tentar acessar o endereço 200, na verdade ele acessará o endereço 1200 da memória, já que o valor do endereço base será somado ao endereço acessado. Observe na tabela abaixo.

Registrador de Endereço Base		Endereço Referenciado		Endereço Físico
1000	+	200	=	1200
2000	+	200	=	2200
1000000	+	200	=	1000200
1000	+	123	=	1123
2000	+	1200	=	3200
1000000	+	123456	=	1123456

Assim para "relocar dinamicamente" um processo, basta alterar o valor do Registrador de Endereço Base no momento da execução daquele processo, apontando a posição inicial da área da memória principal em que aquele processo será colocado para execução, eliminando a necessidade de modificações no código dos programas que estão em execução.

4.1. Espaço Virtual de Endereçamento

* Memória acessível pelo aplicativo dividida em blocos: **páginas**
* MMU: registro de endereçamento base para cada página
  - Mapeamento direto: todos os registros em zero (Tabela)
  - Sobreposição de áreas (Tabela)
  - Inversão de áreas (Tabela)
  - Descotinuidade completa (Tabela)
* Estes dados ficam armazenados para CADA processo
  - Informação de "estado", como visto na unidade anterior
* Diminui x Elimina desfragmentação

O sistema de endereçamento virtual é, na verdade, uma combinação de vários dos recursos anteriormente citados e está presente na maioria dos sistemas operacionais modernos, devendo, para isso, ser suportado pelo hardware.

A idéia é simples e é baseada na idéia do registrador de endereço base, mas bastante mais flexível. Por exemplo, imagine que, ao invés de um registrador de endereço base, existam vários deles. Neste exemplo, é possível existir um registrador para cada bloco de memória. Ou seja: se há 1024 posições de memória, pode-se dividir este conjunto em 8 blocos de 128 bytes de memória, chamdas *páginas*. Cada página terá seu próprio registrador de endereço base:

Posição de Memória Referenciada
0 ~ 127
128 ~ 255
256 ~ 383
384 ~ 511
512 ~ 639
640 ~ 767
768 ~ 895
896 ~ 1023

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0

Com a configuração apresentada acima, existe um "mapeamento direto", ou seja, os valores dos registradores de endereço base são tais que a posição referenciada corresponde exatamente à posição de mesmo endereço na memória física. Entretanto, isso não é

obrigatório: suponha que, por alguma razão, se deseje que quando forem acessados os dados de 128 ~ 255, na verdade sejam acessados os dados de 256 ~ 383. Como fazer isso? Simples: basta alterar o registrador de endereçamento número 1 (da área 128 ~ 255) com o valor 128. Assim, quando o endereço 128 for referenciado, o endereço físico acessado será 128 + 128 = 256. Quando o endereço 255 for referenciado, o endereço físico acessado será 255 + 128 = 383.

Posição de Memória Referenciada
0 ~ 127
128 ~ 255
256 ~ 383
384 ~ 511
512 ~ 639
640 ~ 767
768 ~ 895
896 ~ 1023

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento	Posição de Memória Física Acessada
0	0	0 ~ 127
1	128	256 ~ 383
2	0	256 ~ 383
3	0	384 ~ 511
4	0	512 ~ 639
5	0	640 ~ 767
6	0	768 ~ 895
7	0	896 ~ 1023

Pela mesma razão, pode ser interessante que os acessos à região de endereços de 256 ~ 383 fossem mapeados para os endereços físicos 128 ~ 255. Isso pode ser feito indicando no registrador de endereços 2 o valor -128:

Posição de Memória Referenciada
0 ~ 127
128 ~ 255
256 ~ 383
384 ~ 511
512 ~ 639
640 ~ 767
768 ~ 895
896 ~ 1023

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento	Posição de Memória Física Acessada
0	0	0 ~ 127
1	128	256 ~ 383
2	-128	128 ~ 255
3	0	384 ~ 511
4	0	512 ~ 639
5	0	640 ~ 767
6	0	768 ~ 895
7	0	896 ~ 1023

Observe que isso praticamente resolve de forma completa o problema da desfragmentação da memória, permitindo inclusive que um processo seja executado em porções não contíguas de memória: pelo ponto de vista do processo, ele **estará em uma região contígua**. Por exemplo: um programa de 256 bytes poderia ser alocado da seguinte forma:

Posição de Memória Referenciada
0 ~ 127
128 ~ 255

Número do Registrador de Endereçamento	Valor do Registrador de Endereçamento	Posição de Memória Física Acessada
0	256	256 ~ 383
1	512	640 ~ 767

Para o processo, ele está em uma região contígua da memória principal, dos endereços 0 ao 255. Entretanto, na prática, ele está parte alocado na região 256 ~ 383 e 640 ~ 767. Este mecanismo se chama *espaço de endereçamento virtual com mapeamento de páginas*.

Entretanto, cada processo precisa possuir uma tabela destas. E, de fato, esta tabela faz parte das informações do processo, dados estes armazenados na região de configuração de memória.

A memória virtual permite que a relocação de processos na memória seja simplificada (basta trocar os valores dos registradores de endereço) e diminui muito a necessidade de desfragmentação da memória. Mas ... porque diminui a necessidade ao invés de "elimina"?

A resposta é simples: por questões de economia de espaço, em geral se define que cada processo pode estar fragmentado em até X partes, sendo X um número fixo e, normalmente, não muito alto. Quanto maior for o número de fragmentos permitidos, maior é a tabela que deve ser armazenada por processo e maior é o desperdício de memória com estas informações. Se ocorrer uma situação em que um processo só caberia na memória se estivesse fragmentado em um número de fragmentos maior que X, então será necessário que o sistema desfragmente a memória. De qualquer forma, a desfragmentação fica bem mais rápida, já que a relocação de processos pode ser feita de forma simples, sem mudanças nos programas.

5. MEMÓRIA VIRTUAL (OPCIONAL)

* Mapear em páginas regiões além da RAM física
    - Essas regiões ficam no disco: swapping
* Quando aplicativo tenta acessar uma dessas páginas
    - Sistema detecta
    - Traz página para a memória física
    - Ajusta registradores de endereço de página
    - Permite o acesso
* Processo transparente para o software
    - Perceptível para o usuário
        + Lentidão
        + *Thrashing*

Com base na idéia do sistema de endereçamento virtual apresentado anteriormente, muitos processadores e sistemas operacionais implementaram suporte para o que se convencionou chamar de "memória virtual", que nada mais é do que a incorporação da tecnologia de *swapping* em conjunto com o endereçamento virtual. O resultado é como se a memória total disponível para os aplicativos fosse a memória RAM principal física **mais** o espaço disponível na memória secundária. Como isso é feito?

Considere que um sistema tem 1GB de memória principal física (RAM) e vários GBs de disco. A idéia é que um registrador de endereçamento base **possa apontar posições de memória maiores do que a memória física**. Para o processo, ele é acarregado como se o sistema de fato tivesse mais memória disponível do que a física. Toda a parte do processo que ultrapassar a memória física disponível, será automaticamente jogada para o arquivo de troca (swap). Em outras palavras, parte do processo será colocada no disco.

Entretanto, quando o processo tentar acessar uma destas regiões que estão além da memória física, isso disparará uma ação do sistema operacional que irá recuperar do arquivo de swap o bloco que contém a informação desejada, colocando-a de volta na memória principal e corrigindo o registrador de endereçamento virtual para refletir a nova realidade e, assim, permitindo que o programa acesse aquela informação. Note que tudo isso ocorre de maneira absolutamente transparente para o processo.

Esta técnica permite o swapping de forma 100% transparente ao processo, facilitando muito o funcionamento do sistema operacional e mesmo a programação dos processos. O sistema acaba se comportando, realmente, como se a quantidade de memória disponível fosse bem maior. Entretanto, existe uma penalidade: como a memória secundária é, em geral, bastante mais lenta que a memória principal, o desempenho do equipamento como um todo é bastante prejudicado quando o uso de memória virtual (swap) é necessário.

Quando a atividade de swap cresce muito e o sistema se torna muito lento por causa disso, dá-se o nome de *thrashing* do sistema.

6. BIBLIOGRAFIA

DAVIS, W.S. Sistemas Operacionais: uma visão sistêmica. São Paulo: Ed. Campus, 1991.

MACHADO, F. B; MAIA, L. P. Arquitetura de Sistemas Operacionais. 4ª. Ed. São Paulo: LTC, 2007.

SILBERSCHATZ, A; GALVIN, P. B. Sistemas operacionais: conceitos. São Paulo: Prentice Hall, 2000.

TANENBAUM, A. S. Sistemas Operacionais Modernos. 2ª.Ed. S. Paulo: Prentice Hall, 2003.

Unidade 5: Gerência de Dispositivos

Prof. Daniel Caetano

Objetivo: Apresentar o conceitos envolvidos na gerência de dispositivos e sua influência no software.

Bibliografia: MACHADO e MAIA; TANENBAUM; SILBERSCHATZ e GALVIN.

INTRODUÇÃO

Conceitos Chave:

- Problema: Desenvolver software aproveitando recurso especial do hardware
  * Desenvolver o *driver*!

Certo dia, desenvolvendo software para um celular de último tipo, surge pela primeira vez o seguinte problema: os requisitos do sistema sendo desenvolvidos exigem o uso de um dispositivo que *está disponível no hardware*, mas que está inacessível pelo sistema operacional do celular.

Uma vez que "desprezar o recurso" não é uma opção válida, ao consultar o superior vem a constatação: "o cliente quer que o *driver* seja também desenvolvido". Mas o que é o driver? O que ele tem a ver com o dispositivo? Por que ele é necessário?

Alguns destes assuntos serão apresentados nesta aula.

1. DISPOSITIVOS

- O que são dispositivos?
  => Equipamentos de Entrada e Saída
  * Teclado, monitor, caixa de som... x placa de vídeo, IDE, SCSI...
- Dispositivos Estruturados x Não-Estruturados
  * Disco, fita x Teclado
- Necessidade de suporte => Programar diretamente?

Quando se fala em dispositivo, provavelmente a primeira imagem que vem à cabeça de muitos é uma impressora, um mouse, um teclado... e, de fato, estes são dispositivos. Mas será que todos os dispositivos são visíveis?

É certo que não. Um dispositivo é todo equipamento externo ou interno ligado a um computador que permita operações de "entrada e saída", isto é, interação com o usuário e/ou armazenamento/recuperação de dados. Neste sentido, caixas de som, discos rígidos, monitor de vídeo... todos são dispositivos.

Os dispositivos podem ser *estruturados* ou *não-estruturados*. Os dispositivos estruturados são aqueles que as informações são armazenadas em blocos de tamanhos fixos, e o acesso a elas pode ser aleatório (unidade de disco) ou seqüencial (unidade de fita DAT). Já os dispositivos não-estruturados são aqueles que as informações não são disponíveis em blocos fixos e, assim, seu acesso pode ser apenas seqüencial (teclado).

Convém lembrar que, normalmente, a memória RAM e a CPU não são considerados dispositivos. Também as IDEs, SCSIs, USBs e outros chamados *controladores* não são, em geral, considerados dispositivos, no sentido sugerido anteriormente.

Assim, se o celular a ser programado possui um dispositivo de áudio não suportado pelo sistema operacional do mesmo, o software deverá ser programado para isso. Mas essa programação terá de ser feita diretamente? Será necessário entender como produzir um som pela caixa de som? Não necessariamente.

1.1. Controladores de Dispositivo

- Hardware de Interface
  - * IDE, SCSI, Placa de Vídeo / Som
  - * Registradores, Buffers, DMA...
  - * Controle de Erros
- Programar os controladores diretamente? => Subsistema de E/S!

A maioria dos dispositivos possui um *controlador*, um hardware denominado *interface,* cuja função é facilitar a programação do dispositivo. Um controlador tem registradores e comandos próprios, além de cuidar dos eventuais sistemas de *buffer* (dispositivos como teclado) e transferência para a memória com DMA (como ocorre no controlador de disco, placa de som...), quando necessário.

Desta forma, para programar um dispositivo, é necessário programar o seu *controlador*. Assim, será necessária a documentação técnica de controlador para poder realizar esta programação.

Exemplos de controladores clássicos são as IDEs, as SCSIs, as USBs, as placas de vídeo, placas de som etc.

No caso do dispositivo de áudio, se alguns dados corretos forem enviados para a placa de som, ela se encarregará do resto, ou seja, de produzir o som usando as caixas de som, facilitando muito a vida do programador.

Mas, então, o desenvolvimento visará a acessar diretamente estes controladores de dispositivo? É possível, mas é importante lembrar que o sistema operacional já oferece vários recursos que, se o programador puder usar, serão de muito interesse.

Por exemplo: se o objetivo for desenvolver um programa que acesse a placa de som cheia de recursos do novo modelo de celular, mas o programa estiver sendo executado em uma versão mais antiga do celular, que tem uma placa de som mais simples, o resultado pode não ser o mais adequado quando se faz um software que acessa diretamente o controlador do dispositivo. Assim, se for possível tirar proveito do *subsistema de entrada e saída (E/S)* do sistema operacional, tanto quanto melhor.

2. SUBSISTEMA DE E/S

- O que faz subsistema de E/S => Gerenciar Dispositivos!
  * Recebe requisição => envia ao controlador de dispositivo correto.
- *Independência de Dispositivos*
  * Gravar arquivo... pen-drive? disco? rede?
- Subsistema de E/S x Controladores de Dispositivo
  * Necessidade de tradutor => Driver

O subsistema de E/S é uma parte do sistema operacional responsável por gerenciar os acessos aos controladores de dispositivo. Sua função é, basicamente, receber comandos complexos (como, por exemplo: "toque este vídeo") dos programas aplicativo e *redirecioná-lo ao controlador de dispositivo correto*.

Apesar de parecer um conceito simples, é este conceito que permite a *independência de dispositivos* de um programa aplicativo; o software pode escrever em uma unidade qualquer, sem ter de se preocupar se é uma unidade de rede, disco, pen-drive ou fita. O subsistema de E/S fará este direcionamento automaticamente.

Em outras palavras, quando um aplicativo deseja escrever um dado em um dispositivo, ele simplesmente faz a solicitação ao subsistema de E/S, e este se encarrega de entregar a solicitação ao controlador de dispositivo adequado.

Mas será que isso não traz um problema? Será que todos os controladores de dispositivos "falam a mesma língua"? Será um controlador VGA tem os mesmos comandos que um controlador de som?

A resposta é, obviamente, não. Como este tipo de subsistema pode funcionar, então? A resposta está em um outro pedacinho de software: os *device drivers.*

3. DEVICE DRIVERS

- O que é um device driver? => Tradutor!
  - * Disco: bloco => disco, setor...
  - * Som: áudio bruto => sinais elétricos para caixa de som

Até o momento foi apresentado que, para programar um dispositivo, a interação é, na realidade, feita com o controlador do dispositivo. Adicionalmente, deseja-se poder usar o subsistema de E/S do sistema operacional, a fim de tirar proveito da "independência de dispositivos".

Entretanto, também foi apresentado que nem todos os controladores de dispositivo "falam a mesma língua", algo que seria necessário para que o subsistema de E/S pudesse operá-los. Ora, como o subsistema de E/S sempre fala "a mesma língua" mas não os controladores de dispositivo, a solução está em criar um *tradutor*, um trecho de software intermediário que entenda as solicitações do controla. O nome deste tradutor é *device driver*.

Assim, a função básica do *device driver* é receber solicitações do subsistema de E/S e *traduzi-las* para o controlador de dispositivo. Por exemplo: o device driver de disco recebe a solicitação para ler um bloco do disco e transforma isso em uma série de comandos para o correto posicionamento da cabeça de leitura do disco, além de verificar por erros de leitura. Um outro exemplo: o device driver da placa de som recebe a instrução para tocar um arquivo de som, e converte isso em uma série de comandos para seus dispositivos controlarem a conversão e envio de dados para a caixa de som.

Uma outra função importante dos device drivers é o controle de acesso aos dispositivos. Alguns dispositivos permitem acesso por múltiplos agentes simultâneos, outros não.

3.1. Algoritmos de Controle de Acesso

- Controle de Acesso
  - * Acesso Simultâneo => Placa de Som (Não Estruturado)
  - * Acesso Alternado => Disco (Estruturado)
  - * Acesso Exclusivo => Impressora (Não Estruturado)
- Subsistema x Controle de Acesso
  - * Acesso Exclusivo
    - = Spool => Impressora
    - = Janelas => Vídeo

Acesso Simultâneo: alguns dispositivos, como algumas placas de som, permitem acesso simultâneo de diversas aplicações. Os dados podem ser reproduzidos por canais diferentes (ex.: placas de som com múltiplos canais) ou mesmo serem mixados (ex.: placas de som um único canal).

Acesso Alternado: alguns dispositivos, como controladores de disco, permitem acesso alternado de diversas aplicações. Isto significa que diferentes aplicações podem escrever, alternadamente, no disco, ainda que nenhuma delas tenha terminado de escrever tudo. Cada uma escreve no ponto que precisa ser escrito.

Acesso Exclusivo: alguns dispositivos, como impressoras, permitem acesso exclusivo de uma aplicação. Isto significa que uma única aplicação pode usar o dispositivo e ele só será liberado ao fim do uso.

3.1.1. Papel do Subsistema de E/S no Controle de Acesso

No caso de dispositivos de acesso exclusivo (como impressoras e vídeo, por exemplo), é comum que o subsistema de E/S faça algum tipo de gerenciamento para que o acesso exclusivo não represente uma limitação ao uso do equipamento.

Por exemplo, no caso das impressoras, é comum que o subsistema de e/s tenha, para a impressão, um sistema de fila de impressão (chamado de *spool*), que organizará os documentos para impressão na ordem em que chegaram, para que sua impressão ocorra uma a uma, sem bloquear o computador de quem solicitou a impressão.

O mesmo costuma ocorrer com o subsistema de vídeo em sistemas operacionais multitarefa gráficos, como o Windows e MacOS. Como o acesso simultâneo direto da tela por todas as aplicações pode ser problemático (uma sobrescrevendo a outra), o subsistema de vídeo fica responsável por gerenciar as áreas onde um aplicativo pode ou não escrever, simulando as *janelas* de impressão em vídeo.

3.2. Recursos de Comunicação com Dispositivos

- Recursos de Comunicação com Dispositivos
    * Portas de E/S
    * Endereços de Memória
    * Interrupções
    * Canais DMA

Um computador tem, em geral, quatro tipos de recursos que são utilizados por dispositivos, de forma que os programas e drivers possam se comunicar com os mesmos. Estes recursos são, em geral, gerenciados pelos device drivers. Tais recursos de comunicação com os dispositivos de hardware são:

- **Endereçamento de E/S (Portas de E/S)**: esta é a forma tradicional que um software usa para se comunicar com um dispositivo e/ou hardware controlador de dispositivo. Em geral, estas "portas de E/S" são acessadas diretamente apenas pelo Device Driver, mas podem ser acessadas por programas em geral, em alguns casos, através do Sistema Operacional. Tipicamente uma "porta de E/S" é como se fosse o "endereço" de um dispositivo, e dados podem ser enviados para este dispositivo usando uma instrução do tipo ***OUT (porta), dado***.

Por exemplo: se um device driver de placa de vídeo precisa prover uma função "point(x,y,c)" para desenhar um ponto na tela, a placa de vídeo está ligada à porta 80h e o comando para escrever um ponto nesta placa de vídeo é conseguido enviando-se, na seqüência: X, Y, Cor, 20h, a função em C que faria isso seria algo como:

```
void point (int x, int y, int c) {
      inline {
            OUT (80h),x
            OUT (80h),y
            OUT (80h),c
            OUT (80h),20h
            }
      }
```

- **Endereçamento de memória**: Apesar dos endereços de memória serem usados para que o processador leia e escreva na memória, um dispositivo também pode estar ligado a um um endereço de memória. Neste caso, um dado escrito naquele endereço não será gravado na memória, mas sim enviado para o dispositivo. Para isso não se usa o comando *out*, mas sim o comando ***LD (endereço),dado*** (ou, na arquitetura x86, ***MOV [endereço],dado***).

Por exemplo: se um device driver de placa de vídeo precisa prover uma função "point(x,y,c)" para desenhar um ponto na tela, a placa de vídeo está ligada ao endereço de memória 1000h e o comando para escrever um ponto nesta placa de vídeo é conseguido enviando-se, na seqüência: X, Y, Cor, 20h, a função em C que faria isso seria algo como:

```
void point (int x, int y, int c) {
      inline {
            LD (1000h),x
            LD (1000h),y
            LD (1000h),c
            LD (1000h),20h
            }
      }
```

Porque usar um endereço de memória para isso? Bem, em algumas arquiteturas pode nem existir as portas (processadores ARM7 ou ARM9, por exemplo). Aí não há alternativa. De fato, a maioria dos videogames têm todos os seus dispositivos "mapeados em memória".

Há uma outra razão, porém, para se utilizar endereços de memória: quando o dispositivo tem uma memória própria. Apesar da memória de um dispositivo também poder

ser acessada por portas (com algum malabarismo), é muito mais prático acessá-la como se fosse memória do computador principal.

Assim, alguns hardwares que possuem memória "mapeiam" sua memória como se fosse memória principal. Desta forma, se uma placa de vídeo usa os endereços de 0010:0000h a 001F:0000h como sua memória, qualquer dado escrito ou lido naquela região (com a instrução LD/MOV) acessará a memória da placa de vídeo, e não a memória principal.

Como a placa de vídeo usa os dados dessa memória para compor a tela, um byte escrito na memória de vídeo aparece instantaneamente na tela.

- **Interrupções (IRQs)**: Algumas vezes o dispositivo precisa avisar o software que algo aconteceu: a placa de rede precisa avisar que seu buffer está cheio, a placa de som precisa avisar que acabou de tocar uma música, a placa de vídeo precisa avisar que a tela já terminou de ser pintada.

Os dispositivos usam, então, o recurso de interrupção, que nada mais é que um sinal de hardware sobre o qual o processador pode atuar (por exemplo, desviando a execução para um trecho de código específico, com o driver daquele dispositivo).

Como existem vários dispositivos no computador, é interessante ter pelo menos uma interrupção para cada dispositivo. A arquitetura original do PC, por exemplo, tem 16 interrupções, gerenciadas por um processador chamado PIC (Programmable Interrupt Controler).

Assim, se a placa de som usa a interrupção 5, o driver da placa de som avisa ao Sistema Operacional que, quando a interrupção 5 for disparada, ele deve ser avisado. O sistema faz isso chamando uma função do driver, como por exemplo a função chamada *signal*, que deve tratar a informação.

```
void signal (int parametro) {
      // realiza as operações necessárias,
      // de acordo com o parâmetro.
      (...)
      }
```

Entretanto, nem tudo é perfeito e pode ser que mais de um dispositivo esteja *compartilhando a mesma interrupção*. Neste caso, antes de tratar a interrupção é preciso que o driver verifique se ela foi gerada pelo dispositivo que ele controla.

A verificação é trivial, digamos assim: o driver deve verificar se a interrupção foi realmente causada pelo controlador pelo qual ele é responsável e, caso não for, deve avisar o sistema operacional, para que ele continue procurando qual é o controlador de dispositivo que precisa de atenção. Um exemplo de como isso pode ser feito vem a seguir.

```
boolean signal (int parametro) {
      // Verifica controlador se foi ele o causador da IRQ.
      (...)
      // Se não foi...
      if (x==false) return false;
      // Se foi, realiza as ops necessárias, usando o parâmetro.
      (...)
      return true;
      }
```

Infelizmente alguns drivers não tomam este cuidado (ou simplesmente o dispositivo não admite compartilhamento de interrupções) e, portanto, exigem uma *irq exclusiva*. Nestes casos, é claro, o driver avisa o Sistema Operacional disso, para que outros drivers não possam usar esta mesma interrupção.

Entretanto, como o número de interrupções pode ser limitado, isso pode levar a alguns conflitos e fazer com que alguns dispositivos não funcionem corretamente ou simplesmente não funcionem.

Por essa razão, na arquitetura x86, o PIC foi substituído pelo APIC (Advanced PIC), que é capaz de gerenciar até 256 interrupções, das quais 32 são reservadas para uso de software e compatibilidade, e as outras 224 estão disponíveis para os dispositivos.

- **Direct Memory Access (DMA):** Em arquitetura de sistemas é, em geral, apresentado o conceito de DMA, que permite que alguns dispositivos e/ou controladores de dispositivo troquem informações diretamente com a memória do computador, sem a necessidade de auxílio da CPU principal.

Este recurso é provido por um circuito especial, muitas vezes embutido na própria CPU. E este circuito precisa ser programado pelo device driver para executar sua tarefa; da mesma forma que a interrupção, o DMA pode ser compartilhado ou não.

No PC ele não costuma ser compartilhado nos sistemas operacionais e, assim, são necessários vários circuitos de DMA. O número de "canais DMA", como são chamados, varia na arquitetura. Os controladores de DMA também podem ser de 8, 16, 32 ou mais bits. Isso significa que um DMA de 16 bits transfere informações em múltiplos de 16 bits, por exemplo.

4. CONCLUSÃO

Assim, conforme visto, será necessário programar um *device driver* para controlar o dispositivo de áudio do celular; este *device driver* deve estar de acordo não só com os requisitos do controlador de dispositivo, mas também com os requisitos do subsistema de E/S do sistema operacional. Graficamente, pode-se expressar estas relações como um modelo de camadas, conforme a figura 1:

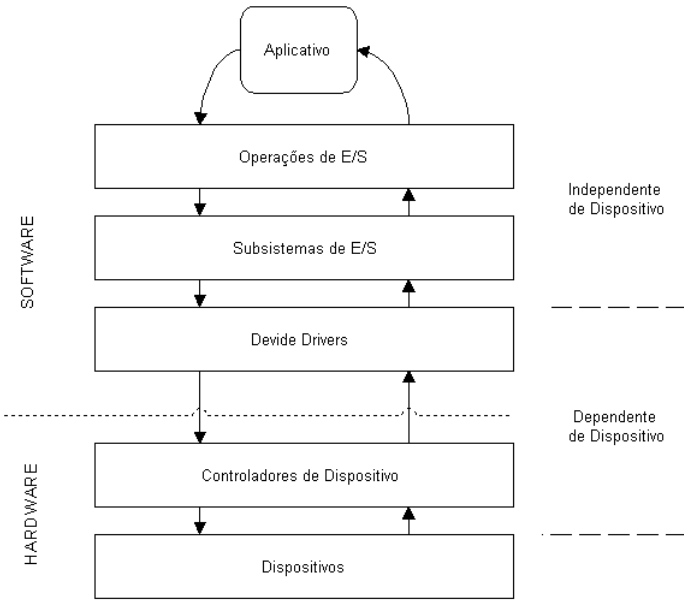


Figura 1: Camadas da Gerência de Dispositivos (baseado em MACHADO e MAIA, 1999)

5. ATIVIDADE

Uma empresa tinha um sistema de medição de temperatura de equipamentos (um hardware) ligado a um computador velho, que funcionava com o sistema operacional Windows 95. Um outro departamento de sua empresa foi contratado para modernizar o sistema, mas o dispositivo de medição de temperatura permanecerá o mesmo.

Segundo o outro departamento de sua empresa, o novo software precisará do Windows 7 e o computador onde é conectado o dispositivo de temperatura será também trocada. Isso garantirá um melhor desempenho do sistema no controle de temperatura das máquinas da fábrica, que é controlado pela rede, através de outros terminais, por meio deste servidor. Outro detalhe que a outra equipe detectou é que o driver disponível para o dispositivo não funciona de forma alguma no Windows 7.

Assim, seu chefe solicitou um relatório que explique a necessidade do desenvolvimento adicional de um "software adaptador", que terá um custo adicional. A empresa questionou a necessidade e insiste em que se use o driver de dispositivo já existente.

Proponha também uma solução para o problema, especificando todos os detalhes que puder da solução, além de informar quais as documentações que serão necessárias ao desenvolvimento.

6. BIBLIOGRAFIA

MACHADO, F. B; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª. Ed. São Paulo: LTC, 2007.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª.Ed. S. Paulo: Prentice Hall, 2003.

SILBERSCHATZ, A; GALVIN, P. B. **Sistemas operacionais**: conceitos. São Paulo: Prentice Hall, 2000.

DAVIS, W.S. **Sistemas Operacionais**: uma visão sistêmica. São Paulo: Ed. Campus, 1991.

Unidade 6: Sistemas de Arquivos

Prof. Daniel Caetano

Objetivo: Apresentar o conceitos dos sistemas de arquivos e a influência do funcionamento dos principais sistemas de arquivos no desempenho dos softwares.

Bibliografia: MACHADO e MAIA; TANENBAUM; SILBERSCHATZ e GALVIN.

INTRODUÇÃO

* Problema: Sistema baseado em banco de dados de desempenho crítico.
    - Qual sistema de arquivos usar?
* Características dos sistemas de arquivos
    - Velocidade
    - Tolerância a Falhas (Recuperação)
    - Compatibilidade
* Organização Lógica x Física

Quase sempre que se desenvolve um sistema de médio ou grande porte, este sistema envolve um servidor principal, com um banco de dados, onde a aplicação será executada, fornecendo dados para todos os outros equipamentos a ela ligados pela rede.

Uma das discussões que poucas vezes é feita, mas que é de grande importância, é a discussão sobre o sistema de arquivos a ser usado neste equipamento. Apesar de parecer uma discussão sem propósito para o leigo, esta decisão é de fundamental importância por definir uma série de características como velocidade, tolerância a falhas, recuperação e compatibilidade.

Para entender a razão de tantas diferenças entre os diferentes sistemas de arquivos, é interessante observar o que ocorre por dentro dos mesmos, sendo este o principal assunto desta unidade.

Inicialmente será feita uma apresentação sobre a parte do sistema de arquivos que o usuário vê, ou seja, a organização lógica em arquivos e diretórios. Em seguida será feita uma apresentação dos conceitos da organização física dos dados no disco.

1. ARQUIVOS

* Conjunto de Informações Relacionadas
  - Programas
  - Dados
    + Não estruturados => texto
    + Estruturados => Banco de Dados
      = Registros => Fixo x Variável
* Acesso Seqüencial x Direto
* Características dos Arquivos
  - Nome
  - Atributos
* Organização? => Diretórios

Todos que lidam com computadores estão acostumados ao conceito de "arquivo", mas o que seria, formalmente, um arquivo?

Um arquivo é, basicamente, um conjunto de informações relacionadas, que podem representar dados ou programas. Um programa é um conjunto de instruções para um processador. Um arquivo de dados, por sua vez, pode ter uma estrutura interna rígida (como em arquivos de banco de dados) ou uma estrutura menos rígida (como um arquivo de texto).

Em qualquer dos casos, entretanto, um arquivo é um conjunto de dados armazenados e indexados em uma mídia qualquer; é função do sistema operacional manter estes dados na forma como foram armazenados, além de "esconder" a forma exata com que isso é feito.

Internamente, o arquivo poderá ser uma simples seqüência de bytes, sem qualquer estrutura rígida, como um arquivo de texto puro ou com uma estrutura muito bem definida, como um arquivo de vídeo *mpeg*.

Num arquivo sem estrutura, os dados são armazenados como uma seqüência simples de bytes. Num arquivo com estrutura, os dados são armazenados em blocos de bytes, sendo que estes blocos chamados de *registros*, que podem ter um tamanho fixo ou variável.

Exemplo:

Arquivo sem estrutura: arquivo de texto.
Objetivo: ler a primeira palavra do segundo parágrafo.
Como realizar: Só lendo byte a byte, até encontrá-la.

Arquivo de estrutura fixa: banco de dados.
Objetivo: ler o nome do cliente número 37.
Como realizar: Basta ler o bloco 37 e acessar seu nome.

No caso de arquivos não-estruturados, o acesso se dá, normalmente, de maneira *seqüencial*, ou seja, inicia-se do primeiro byte e a leitura ocorre seqüencialmente, até o último byte.

Primeiro parágrafo do texto\r\nSegundo parágrafo do texto\r\nTerceiro parágrafo do texto...

No caso de arquivos com estrutura de tamanho fixa, o acesso pode ser seqüencial ou *direto*, isto é, é possível acessar o registro de dados de interesse diretamente, sem a necessidade de ler todos os dados intermediários. Isso só é possível se os blocos forem do mesmo tamanho, pois sabendo o tamanho do bloco e o número do bloco que se deseja ler, basta fazer uma pequena multiplicação para calcular a posição correta de leitura.

0	10	20	30
Bloco 1	Bloco 2	Bloco 3	Bloco 4

No caso de arquivos com estrutura de tamanho variável, cada bloco tem um indicador da posição de início do próximo bloco, de maneira que é necessário ir de bloco em bloco até encontrar o bloco desejado (embora não seja necessário ir de byte em byte, ainda é um processo lento).

0	8	17	34
8, Bloco 1	17, Bloco 2	34, Bloco 3	56, Bloco 4

Assim, dependendo do tipo de arquivo, é possível acessá-los de diferentes formas. Resumidamente:

- Acesso Seqüencial: Leitura Byte a Byte
- Acesso Direto: Leitura direta do registro desejado.

Mas, antes de realizar qualquer leitura... não há algo que precise ser feito? De fato, se existem diversos arquivos no dispositivo de armazenamento, é preciso primeiro saber quais são estes arquivos, e esta diferenciação é feita, normalmente, pelo *nome de arquivo*.

Cada arquivo deve ter um nome, tamanho, e algumas propriedades (hora de criação, permissões de acesso etc.) atribuídas a ele. Estes dados podem estar todos em uma estrutura-índice, chamada *diretório*, ou podem estar dispersos pelo disco, em blocos de controle chamados Inode (Information Node) ou Fnode (File Node).

Independente do local onde o nome do arquivo é atribuído, existe a necessidade deste diretório, de maneira que o sistema operacional possa encontrá-lo sempre que o programador julgar necessário. Mas como é esta organização em diretórios?

2. ORGANIZAÇÃO DOS ARQUIVOS EM DIRETÓRIOS

* Um "arquivo índice"
  - Nome
  - Estrutura de informação do arquivo x Outros atributos
* Estrutura de Diretórios Aninhados
* Diretório Principal (Root / Raiz)

Um diretório é, essencialmente, um índice de arquivos. Ele é responsável por armazenar os nomes dos arquivos, seu local de armazenamento no disco, seu tamanho, sua hora de criação, modificação, permissões de acesso etc., como pode ser visto abaixo:

0	13	17	21
Nome	Tamanho	Data Modificação	Permissões

Com exceção do *nome de arquivo*, que está presente em praticamente todos os sistemas de arquivos, as outras informações variam de sistema para sistema.

Cada sistema de arquivos tem uma estruturação rígida da "tabela" de arquivos, como a exemplificada acima. Isto faz com que seja esta estrutura de dados de diretório que defina algumas características como, por exemplo, tamanho máximo do nome do arquivo.

Mas, sempre que se fala em diretórios, surge a dúvida: mas afinal, onde ficam estas informações, no dispositivo de armazenamento? A resposta pode parecer um tanto estranha: **em qualquer lugar**! Como assim? Bem, o diretório nada mais é do que um arquivo, ou seja, é um arquivo que indica outros arquivos. Estes arquivos endereçados por ele podem ser arquivos de dados normais ou outros diretórios, criando assim uma estrutura de diretório aninhada:

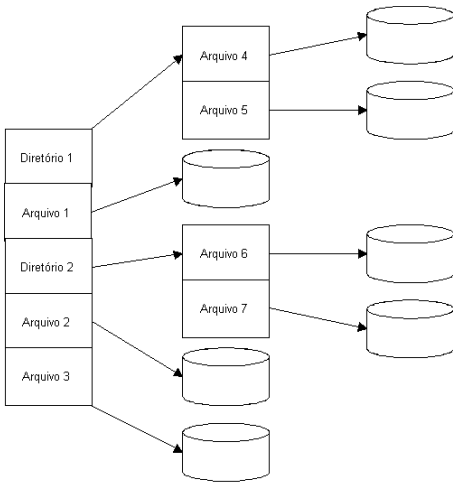


Figura 1: Estruturação de Diretórios Aninhados

Mas, ainda que os outros diretórios sejam arquivos referenciados pelo diretório principal, surge a pergunta: e o diretório principal, onde fica?

O diretório principal é um arquivo especial, usualmente de tamanho fixo, e que fica em uma posição fixa do disco, *fora do sistema de arquivos*. Ele é ponto inicial de busca pelo sistema operacional e seu posicionamento precisa ser bem definido.

3. ESPAÇO EM DISCO

* Organização Lógica dos Dados x Organização Física dos Dados
* Organização => Velocidade e Robustez

Já foram apresentadas as estruturas lógicas de organização dos arquivos, tais como são vistas pelo usuário do sistema operacional e pelos programadores. De uma forma geral, estas partes "visíveis" do sistema de arquivos são similares em todos os sistemas de arquivos e o acesso aos arquivos se dá apenas através de seu conhecimento.

Entretanto, esta organização lógica quase nada tem a ver com a organização física dos dados no disco, ou seja, os reais locais em que as informações estão e como elas estão distribuídas. Apesar de não ter uma grande influência na programação, esta organização é de fundamental importância pois ela é quem define a robustez, velocidade, dentre tantas outras características possíveis a um sistema de arquivos.

3.1. Identificação de Espaços Livres

* Maneiras Diferentes de Especificar
    - Mapas de Bits
        + Bloco de Armazenamento (cluster)
        + Gasta espaço Fixo
            = Nota sobre "gigabyte de HD"
        + Busca de espaço vazio rápida
        + Contagem rápida
    - Listas Encadeadas
        + Indicação de tamanho e ponteiro nos blocos livres
        + Gasta espaço variável
        + Busca de espaço vazio lenta
        + Contagem lenta

O primeiro aspecto importante quando se fala na organização física dos dados no disco é a organização do espaço livre. A razão para isto é que, além de saber onde os arquivos

estão no disco (posição esta indicada na estrutura do diretório), quando se pretende criar um novo arquivo é necessário saber *onde há espaço livre* para que ele seja colocado.

Há diversas formas de realizar esta tarefa, sendo as mais comuns o uso de *mapas de bits* e *listas encadeadas*.

3.1.1. Mapas de Bits

A idéia dos mapas de bits é ter uma região do disco dedicada a armazenar a informação sobre quais partes do disco estão livres e quais não. Como 1 bit é suficiente para indicar "sim" (1) ou "não" (0), usa-se um bit para cada *bloco de armazenamento*.

Um bloco de armazenamento - normalmente chamado de *cluster*, é a menor unidade de armazenamento de um sistema de arquivos, podendo ser de 1 byte a até dezenas de quilobytes. Os tamanhos usuais variam de 512 bytes a 4096 bytes (estes valores estão, usualmente, ligados a limitações dos dispositivos de armazenamento ou aos tamanhos usuais de arquivos usados).

Não se usa blocos muito pequenos porque, se assim fosse, seria um enorme desperdício de espaço em disco com o mapa de bits de registro de espaço livre: 1 bit para cada byte = 1 byte para 8 bytes = 1 mega para 8 megas = 1 giga para 8 gigas.

Da mesma forma, não se usam blocos muito grandes, porque um bloco é o mínimo que um arquivo irá ocupar no disco. Assim, se um bloco tivesse (exageradamente) 1MB, um disco com 1GB poderia armazenar no máximo 1000 arquivos, fossem eles de 1 byte ou de 1MB cada.

ATENÇÃO: Apesar de 1KB ser 1024 bytes, 1MB ser 1024 KB, 1GB ser 1024 MB, essa **NÃO É** a convenção usada pelos fabricantes de HD, que usam múltiplos de 1000, ao invés de 1024. Assim, quando cmopramos um HD de "500GB", na verdade recebemos um HD de cerca de 466GB.

NOTA: Um HD de 900GB com blocos de 1 byte, teria 800GB de espaço livre e 100GB de tabela de bits de espaço livre. Neste caso, há desperdício com a tabela.

Um HD de 900GB com um bloco de 1GB, permitiria apenas 900 arquivos de 1 byte e já indicaria "disco cheio". Neste caso, há desperdício nos blocos de dados armazenados.

Neste sistema, para saber se um espaço está livre, basta verificar o mapa de bits: onde houver uma região de "zeros", é uma área livre do disco.

Para saber o espaço livre total, basta contar os "zeros" do mapa de bits e multiplicar pelo tamanho do bloco, algo que é feito de maneira razoavelmente rápida, em especial porque os sistemas operacionais mantém, normalmente, uma cópia deste mapa na RAM (outra razão para que ele não seja muito grande).

3.1.2. Listas Encadeadas

Como os mapas de bits supostamente desperdiçam espaço, desperdício este que depende do tamanho do bloco, algumas propostas surgiram para contornar o problema, sendo a mais comum a de Listas Encadeadas de espaços vazios.

Neste esquema, há uma entrada no diretório principal que indica o primeiro espaço vazio em disco e, neste espaço vazio, há dois dados úteis:

1) Seu comprimento;
2) Indicação (endereço em disco) do próximo espaço vazio.

0	10	27	35	55	87
Dados	17, 35	Dados	20, 87	Dados	13, xx

A vantagem deste método é que só se gasta espaço armazenando dados sobre espaço vazio *se existir espaço vazio*. Adicionalmente, isso quase que elimina o problema dos tamanhos de blocos: a única restrição é que o espaço vazio nunca pode ser menor do que o necessário para relatar as informações descritas acima (ou seja: alguns bytes).

Por outro lado, sempre que for necessário escrever um novo dado, o sistema precisará buscar um local para armazenar o arquivo por toda esta lista, de forma seqüencial e com muitas leituras de disco.

O mesmo problema surge na hora de determinar o espaço em disco. Além disso, Não há como, neste caso, manter tal lista na memória, dado que ela tende a ser bastante grande (quanto mais fragmentado estiver o espaço em disco).

3.2. Armazenamento dos Dados

* Difrentes Formas de Armazenar Dados
    - Armazenamento Contíguo
        + Rápido
        + Informação de local apenas no diretório
        + Fragmentação de Vazios => Problema!
        + Permite acesso seqüencial ou direto
    - Armazenamento por Lista Encadeada
        + Lento
        + Informações de local no diretório e espalhadas pelo disco
        + Permite armazenamento fragmentado
        + Apenas acesso seqüencial
    - Armazenamento Indexado
        + Bloco de Armazenamento (cluster)
        + Rápido

+ Informações de local no diretório e na tabela de indexação
+ Permite armazenamento fragmentado
+ Permite acesso seqüencial e direto

Uma vez determinado o local em que o arquivo será colocado, por estratégias similares às do armazenamento em RAM (*best-fit*, *worst-fit*, *first-fit*), existe a necessidade de armazenar o arquivo, o que pode ser feita de forma *contígua*, *encadeada* ou *indexada*.

3.2.1. Armazenamento Contíguo

O armazenamento contíguo é feito da seguinte forma: no diretório existe a indicação da posição do primeiro byte do arquivo e seu comprimento, e ele estará exatamente naquela posição do disco. Assim, se o diretório descreve:

Nome	**Posição Inicial**	**Tamanho**
Arquivo1.doc	1.000.001	123 bytes

Este arquivo irá ocupar as posições de 1.000.001 a 1.000.123.

Não parece haver qualquer problema com isso, exceto o fato que o tamanho dos arquivos costuma crescer (de fato, é incomum se saber o tamanho de um arquivo no momento em que ele é criado) e isso pode vir a ser um problema, mesmo que exista espaço em disco disponível. Observe a situação abaixo:

Nome	**Posição Inicial**	**Tamanho**
Arquivo1.doc	1.000.001	123 bytes
Arquivo2.exe	1.000.124	100 bytes

Como é possível ver, o espaço logo após o Arquivo1.doc está ocupado pelo Arquivo2.exe. Isso significa que, se o Arquivo1.doc tiver de ser aumentado, isso implicará em uma de três soluções:

a) Desfragmentar, movendo o Arquivo1.doc para outro lugar;
b) Desfragmentar, mover o Arquivo2.exe para outro lugar;
c) Não aumentar o Arquivo1.doc.

É claro que, se há espaço em disco, a solução (c) é inaceitável. Assim, resta a opção de mover os arquivos 1 e 2 para outro lugar... mas e se não houver nenhuma região de tamanho suficiente no disco (apesar de a soma dos pequenos espaços serem mais que suficientes)? Neste caso será necessário realizar uma reorganização geral dos arquivos, chamada *desframentação*, de forma a juntar todos os espaços livres, para que o arquivo possa crescer.

Infelizmente esta opção, que envolve a *defragmentação do espaço livre* em disco, é custosa demais. Assim, este sistema simples de armazenagem, em geral, não é usado.

3.2.2. Armazenamento por Lista Encadeada

O princípio do armazenamento por lista encadeada é similar ao sistema de identificação de espaço livre por lista encadeada; a diferença aqui é que são os blocos de dados que possuem indicação de seu tamanho e onde está o próximo bloco de dado. Será tomada a mesma situação apresentada anteriormente:

Nome	**Posição Inicial**	**Tamanho**
Arquivo1.doc	1.000.001	323 bytes
Arquivo2.exe	1.000.124	100 bytes

1.000.001	1.000.124	1.000.224
123, 1.000.224 Arquivo1.doc	100, xx Arquivo2.exe	200, xx Arquivo1.doc (cont)

Assim, ao ler o Arquivo1.doc, o sistema saberá que, depois do bloco indicado no diretório, é necessário ler o bloco a partir de 1.000.224, onde estará indicado que aquele é o último bloco do arquivo (xx). Este tipo de alocação causa *fragmentação de arquivo*.

Este sistema é muito interessante, apesar de desperdiçar espaço dos blocos de dados com informações de controle; de qualquer forma, ele tem uma caraterística bastante ruim: impede o acesso direto real, já que todo arquivo fragmentado terá de ser percorrido seqüencialmente até que um determinado bloco seja encontrado.

3.2.3. Armazenamento por Indexação

O princípio do armazenamento por indexação é a existência de uma tabela para cada arquivo, onde são indicados os *registros* do dispositivo de armazenamento em que seus dados estão presentes. Assim, considerando um bloco de 128 bytes, é possível descrever a seguinte situação:

Nome	**Blocos**	**Tamanho**
Arquivo1.doc	0, 2, 3	323 bytes
Arquivo2.exe	1	100 bytes

0	128	256	384
Arquivo1.doc	Arquivo2.exe	Arquivo1.doc	Arquivo1.doc

Este esquema permite que o acesso direto seja feito pois, sabendo a posição que se deseja acessar, pode-se calcular qual bloco daquele arquivo deve-se acessar e, com o número do bloco, qual posição do disco deve ser lida.

A desvantagem deste modo de armazenamento é que é necessário um espaço para a tabela de seqüência de blocos de cada arquivo, além de exigir que o disco use um esquema de armazenamento em blocos (com os problemas já citados).

Esta tabela de blocos de arquivo pode estar no dado do diretório, pode ser específica por arquivo (em um bloco de controle) ou ser em separado, da mesma forma que seria feito o mapa de bits de espaço livre, como ocorre no sistema de arquivos FAT.

De fato, no sistema de arquivos FAT a própria tabela de blocos, chamada de "File Allocation Table", indica os blocos que um arquivo usa e é o "mapa de bits" simultaneamente. Isso é feito da seguinte forma: o diretório aponta o bloco incial na tabela, e cada bloco aponta o número do próximo bloco. O último bloco sempre indica o número -1. Blocos com número 0 são blocos livres.

4. CONTROLE DE ACESSO

* Controle de Acesso em Sistema de Arquivos
* Passwords
  - Solicitação Constante
  - Qualquer um pode acessar (password sempre igual)
  - Password permite fazer qualquer coisa com o arquivo
* Grupos de Acesso
  - Não é muito flexível
* ACLs
  - Lento

Muitos sistemas de arquivos fornecem "controle de acessos em nível de sistema de arquivos". Este controle de acesso é feito com informações existentes no diretório, no bloco de controle do arquivo ou mesmo em arquivos de informação de controle separados. A seguir são descritos os principais sistemas de controle de acesso.

4.1. Passwords

A forma mais simples de realizar este controle é através de passwords de arquivo. Os passwords permitem que qualquer pessoa que os conheça, possa acessar um dado arquivo. Este sistema é simples e eficaz, mas tem três problemas básicos:

1) Sempre que o arquivo precisar ser acessado, o password será solicitado;
2) Não há como controlar exatamente "quem" acessou ou alterou um arquivo;
3) Quem possui o password pode fazer o que bem entender com o arquivo.

4.2. Grupos de Acesso

A forma mais comum de controle de acesso, em especial no mundo Unix, é através do sistema "Criador/Grupo/Outros", onde são indicados privilégios de acesso diferenciados para cada uma destas três categorias, em especial quem pode ler, escrever e/ou executar um determinado arquivo.

É possível especificar, separadamente, os privilégios de cada tipo de usuário: para o criador, para os membros de um dado grupo de trabalho e, finalmente, para todas as outras pessoas. É possível definir, por exemplo, que seu proprietário tem todas as permissões, os usuários do mesmo grupo podem ler, mas não escrever ou executar e, finalmente, que este arquivo é completamente inacessível para outras pessoas.

Este esquema resolve os três problemas do password, mas ainda não há muita flexibilidade sobre as permissões. Por exemplo: se uma outra pessoa precisar ter acesso completo ao arquivo, no exemplo apresentado, isso não será possível sem liberar o acesso ao grupo ou a todos.

4.2. Lista de Controle de Acesso

As Listas de Controle de Acesso (Access Control List - ACLs) são a forma tradicional de permitir acesso a arquivos quando muitas permissões diferentes são necessárias a pessoas distintas. A ACL nada mais é que um arquivo (protegido pelo sistema de arquivos) que contém o nome das pessoas cadastradas para acessar aquele arquivo e quais são suas permissões.

É o sistema mais flexível - e também o mais lento - para se proporcionar controle de acessos aos arquivos em nível de sistema de arquivos.

5. CACHE (OPCIONAL)

* Disco => Dispositivo lento
* Estruturas de controle de acesso enormes
* CACHE: armazenamento de dados temporariamente em RAM
* Armazenar estruturas em RAM: bom
* Armazenar dados mais requisitados: bom
    - Política de limpeza
* Lazy Write
    - Falha de Energia

Como é possível perceber, o acesso a arquivos em um dispositivo de armazenamento envolve muitas estruturas de dados, além da lentidão de acesso aos dispositivos físicos. Isso tudo pode ser muito lento.

Para mitigar o problema da lentidão de acesso, é comum que os subsistemas que implementam o sistema de arquivos em um sistema operacional implementem um esquema de armazenamento em memória RAM das informações mais usadas do disco. Este sistema de armazenamento em RAM é chamado de *cache de disco*.

O cache de disco funciona como um pequeno banco de dados, onde os dados lidos recentemente são armazenados. Quando este espaço lota, é necessário "limpar" alguma coisa deste banco de dados; o sistema então "joga fora" o dado mais velho e menos usado da lista, colocando o novo dado por lá.

Se o cache for pequeno, esta operação será feita com muita freqüência, degradando a performance... por outro lado, quanto maior for o cache de disco, mais rápido será o acesso a informações que já foram acessadas anteriormente.

Além do cache de leitura, existe também o cache de escrita. O cache de escrita, também chamado de "*lazy write*" armazena na RAM o conteúdo da escrita, permitindo que o sistema operacional escolha o momento mais adequado para transferir aquelas informações para o disco , quando o disco não estiver sendo utilizado por outras razões, por exemplo.

Isso é positivo e aumenta a eficiência do uso de disco; por outro lado, se houver uma falha de energia de algum tipo, os dados em memória terão sido perdidos e o sistema de arquivos pode ficar em um estado indefinido. Por esta razão a maioria dos sistemas atuais habilita automaticamente a checagem de disco quando há falha de energia.

Ainda pela existência do Lazy Write é que é necessário executar o "eject" dos *pen-drives*, para obrigar o sistema operacional a escrever os dados do cache de escrita no dispositivo, evitando a perda de informações.

6. JOURNALING (OPCIONAL)

* Como funciona uma checagem de disco
* Porque checar tudo?
    - Só o que está aberto?
* Journal
    - Não tem lazy write no journal

Quando há uma falha de energia, toda a estrutura de dados do disco precisa ser checada, para corrigir eventuais problemas (como um arquivo com tamanho errado, já que a sua parte final poderia ainda estar em cache de escrita quando o computador foi reiniciado).

Na prática, o sistema sai verificando a integridade da estrutura de armazenamento (diretórios, tabelas de espaço, tabelas de indexação etc.).

Assim, quanto maior o disco e quanto mais cheio ele estiver, mais complexas são as estruturas a serem checadas. Por esta razão, um sistema com uma unidade de disco com centenas de gigabytes quase cheia, ao ser desligado incorretamente, pode chegar a demorar algumas dezenas de minutos para ser verificado.

Entretanto, alguns desenvolvedores perceberam um fato muito importante: por razões óbvias, os problemas só ocorriam em arquivos que estavam abertos no momento em que o sistema operacional foi reiniciado. Então, no fundo, não era necessário checar o disco todo, no caso de uma falha de memória: bastaria checar os arquivos que estavam abertos.

Para fazer isso, seria necessário criar uma espécie de registro, no próprio disco, indicando todos os arquivos abertos em um determinado momento. Adicionalmente, este registro não pode estar sujeito às regras de *lazy write*, já que nunca pode estar desatualizado.

Este tipo de registro que, assim como as tabelas de espaço livre, não fica na área de espaço de arquivos, foi denominado *journaling*.

O journaling (palavra que se pode traduzir como "diário") torna os processos de abertura e fechamento de arquivo um pouco mais lentos, além de ocupar espaço adicional em disco (reduzindo a região útil). Por outro lado, no caso de falha, um disco de centenas de gigabytes pode ser verificado em alguns segundos.

7. SISTEMAS DE ARQUIVOS MAIS COMUNS

Conceitos Chave:

* FAT/FAT32 (Digital Research / Microsoft)
    - Um dos primeiros sistemas de arquivos
    - Criado para disquete
    - Informações centralizadas => problema!
    - Baixa recuperabilidade
    - Baixa velocidade
    - Alta compatibilidade
* EXT2/3 (comunidade Unix)
    - Bastante antigo => mundo Unix
    - Criado com grandes repositórios em mente
    - Número de arquivos limitado
    - Boa recuperabilidade
    - Velocidade de acesso média/alta
    - Baixa compatibilidade

* HPFS/NTFS (Microsoft / IBM)
    - Fim da década de 1980
    - Melhoria da FAT => para discos grandes
    - Muito boa recuperabilidade (NTFS recentes: excelente!)
    - Alta velocidade de acesso
    - Média compatibilidade
* JFS (IBM) / ReiserFS
    - Fim da década de 1990
    - Uma grande composição de características
    - Disk Span
    - Excelente recuperabilidade
    - Alta velocidade de acesso
    - Baixa compatibilidade

Ao longo dos tempos, diversos sistemas de arquivos foram desenvolvidos - e ainda hoje o são. Os diferentes sistemas de arquivos são, normalmente, voltados a aplicações distintas e, muitas vezes, refletem a realidade da época em que foram criados.

7.1. FAT / VFAT / FAT32

O FAT (File Allocation Table) foi uma das primeiras estruturas de formatação de discos, e sua criação remonta à década de 1970.

O sistema FAT possui duas estruturas básicas de controle: o Diretório Principal, que é o diretório inicial a partir do qual todos os outros diretórios e arquivos são acessados, e a Tabela de Alocação. A tabela de alocação é usada, ao mesmo tempo, como bitmap de áreas disponíveis e tabela de indexação de blocos de disco, chamados *clusters*. Os clusters podem variar de 512 bytes a 64 Kbytes, dependendo do tipo de FAT e do tamanho do disco.

O número no nome da FAT (FAT12, 16, 32) indica o número de bits de cada entrada na tabela de alocação de clusters. Assim, em teoria, o tamanho máximo de uma partição pode ir de 2MB (FAT12 com cluster de 512 bytes) a 256TB (FAT32 com cluster de 64Kbytes).

A FAT é um sistema bastante simples, criado para discos pequenos, e posteriormente adaptado para discos grandes. Sua tolerância à falhas e sua recuperabilidade é baixa (danos ao diretório e/ou à tabela de alocação são quase sempre irrecuperáveis), sua velocidade é baixa (alta fragmentação, consultas freqüentes à tabela de alocação - que se torna enorme em discos grandes e sempre está localizada no início do disco), não conta com atributos de segurança, não possui journaling (o que pode fazer com que a checagem de disco seja bastante lenta) e causa muita fragmentação de espaço.

Por outro lado, o sistema da FAT é o mais adotado em todos os tipos de dispositivos, tornando-o o mais compatível existente, seu uso sendo possível de câmeras fotográficas a computadores e DVD players.

7.2. EXT 2 / EXT 3

O EXT é o sistema de arquivos padrão dos Unix/Linux em geral. O EXT2 é uma versão melhorada do sistema de arquivos originais do Unix e sua criação remonta à década 1960. Apesar de mais antigo que o FAT, o EXT é um sistema que foi criado com pretensões maiores, pois era voltado a computadores de grande porte.

No EXT, uma unidade de disco é dividida em *grupos* (block groups) de dados. Cada *grupo* tem uma estrutura fundamental constituída de diversos tipos de blocos de controle, além dos blocos de dados.

Os blocos de controle fundamentais são os *inodes*, que indicam as informações de um arquivo, como proprietário, tamanho, hora de criação, além da tabela de indexação dos blocos de dados de um arquivo específico. Além dos inodes, existe um local chamado *superblock*, que contém as informações (agregadas) de espaço livre, inodes usados e inodes livres, dentre outras. Adicionalmente, existe um local chamado *block group descriptor*, que contém um bitmap dos blocos usados e livres, inodes usados e livres, etc.

A estrutura do EXT2 é bastante mais complexa, mas na prática funciona como se o disco fosse dividido em diversos pequenos discos, cada um deles com uma estrutura de controle própria, por tabela de indexação. Todos estes "pequenos discos" estão, entretanto, integrados em um único sistema de diretórios. Isso faz com que se um pedaço do disco for danificado, isso não prejudica os dados de outras partes do disco, tornando-o robusto. Além disso, permite um menor desperdício de espaço em disco e tem maior velocidade de acesso na leitura.

Apesar da maior capacidade de recuperação, a checagem de disco do EXT2 é, entretanto, extremamente mais lenta que da FAT. Por esta razão, foi criada uma versão do EXT com journaling implementado. Esta versão é a chamada EXT3.

Uma das maiores limitações do EXT, entretanto, é o número de inodes. Similar ao que acontece com o número de clusters na FAT, o número de inodes é fixo e, se todos forem usados, ainda que haja espaço em disco, não será mais possível armazenar dados. A velocidade do EXT também tem alguns problemas de desempenho: consome mais CPU à medida que os arquivos vão ficando maiores. Quanto à compatibilidade, o sistema de arquivos EXT é limitado ao mundo Unix, tendo sido portado apenas para alguns outros sistemas operacionais, como o antigo IBM OS/2.

7.3. HPFS / NTFS

O HPFS (High-Performance File System) foi criado pela Microsoft para a IBM, como o novo sistema de arquivos para o sistema operacional OS/2, que sucederia o MS-DOS. O objetivo do HPFS era fornecer um grande desempenho para dispositivos de grande capacidade de armazenamento.

A idéia era fazer com que, ao invés de uma única FAT, o disco possuísse diversas FATs. Assim, seguindo a idéia do EXT, o disco seria usado como se fossem diversos pequenos disco FAT, porém com uma integração de estrutura de diretórios, que é organizada sempre em ordem alfabética. Ainda com base na experiência do Unix, muitas das informações do arquivo que eram guardadas no diretório (como nome do arquivo e outros atributos) foram movidos para uma estrutura chamada FNode, que é muito similar ao INode do EXT.

A organização disposição dos bitmaps de espaço livre e FNodes é de tal forma que as consultas em disco sempre minimizem o tempo de busca.

Para as versões de servidor do OS/2, a Microsoft criou juntamente com a IBM o HPFS386, uma versão estendida que, além de mais rápida, implementava critérios de segurança, com controle de acesso aos arquivos. Posteriormente, quando a Microsoft desfez sua parceria com a IBM e criou seu próprio sistema operacional (Windows NT), modificou o HPFS386 para que a área dos FNodes que era usada para armazenar dados diversos (definidos pelo programador) fosse usada apenas para armazenar dados de permissões de acesso e chamou este novo sistema de arquivos de NTFS (New Technology File System).

O NTFS passou por algumas mudanças internas (nada relativo à sua estrutura), e as versões mais recentes implementam uma espécie de journaling, tornando o mais rápido na recuperação de falhas.

O HPFS e NTFS são sistemas de arquivos relativamente rápidos e com boa recuperabilidade. O HPFS é um sistema de arquivos bastante antigo e bem suportado nos diversos Unixes, eComStation e também está disponível uma versão antiga dele para o Windows NT (até XP). O NTFS é um dos sistemas mais suportados, sendo sua leitura disponível em quase qualquer sistema operacional atual. Ainda há poucos dispositivos que suportam leitura/escrita em NTFS.

7.4. JFS

O Journaling File System (JFS) é de estrutura mais complexa dos sistemas citados. Em grande parte, ela é baseada no EXT, porém com número de INodes dinâmico (são criados à medida do necessário) e incorporando características importantes do HPFS/NTFS. Além disso, ele foi construído desde sua fundação com a eficiência do journaling em mente.

É um dos sistemas mais robustos, de melhor e mais rápida recuperação. As leituras são bastante rápidas, embora a escrita ainda perca para outros sistemas de arquivos. Permite não apenas que uma partição seja ampliada, mas também que diversas partições sejam unidas em uma única partição lógica, mesmo que estejam em meios físicos (HDs) diferentes.

Seu grande diferencial com relação à maioria dos outros sistemas de arquivos é oferecer todos estes benefícios sem consumir muita CPU. Além disso, sua recuperabilidade é alta, o que o torna muito interessante para servidores.

A compatibilidade é, entretanto, restrita. Dos sistemas atuais, existem versões de JFS para eComStation e para diversos Unix. Não há dispositivos de uso geral que suportem este sistema de arquivos e nenhum Windows o reconhece.

8. ATIVIDADE

Sua empresa foi contratada para configurar um sistema em uma empresa; um das decisões que serão necessárias é a escolha de um sistema de arquivos para o computador que armazenará um enorme Banco de Dados, que deve estar online o máximo de tempo possível. Adicionalmente, sabe-se que este banco de dados é usado mais para a adição de informações do que para a consulta. O Banco de Dados sendo usado está disponível para todos os sistemas operacionais.

Sua equipe foi selecionada para escolher este sistema de arquivos, mas antes terá de especificar os requisitos que serão observados para a escolha do mesmo. Quais serão estes requisitos?

9. BIBLIOGRAFIA

MACHADO, F. B; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª. Ed. LTC, 2007.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª.Ed. Prentice Hall, 2003.

SILBERSCHATZ, A; GALVIN, P. B. **Sistemas operacionais**: conceitos. PrenticeHall, 2000.

DAVIS, W.S. **Sistemas Operacionais**: uma visão sistêmica. São Paulo: Ed. Campus, 1991.

Unidade 7: Comunicação entre Processos e Paralelismo

Prof. Daniel Caetano

Objetivo: Apresentar o conceitos da comunicação entre processos no processamento paralelo.

Bibliografia: MACHADO e MAIA; TANENBAUM; SILBERSCHATZ e GALVIN.

INTRODUÇÃO

* Problema:
  - Descompressor e Player de Vídeo em tempo real?
  - Transmissão de dados pela rede?
* Múltiplos Processos
* Vantagens:
  - Múltiplas fontes de dados (formatos de entrada)
  - Aproveitamento de diversas CPUs
  - Simplificação de Sincronia com o Hardware
* Compartilhamento de recursos
  - *Buffer*
  - Necessidade de Sincronia
  - Disputa por recursos

Um problema bastante comum no desenvolvimento de sistemas que usam algoritmos complexos (e, muitas vezes, lentos) é a necessidade de execução em paralelo.

Um caso bastante comum é o processamento de dados transmitidos por rede. Em geral, deseja-se que exista um processo que monitore a rede e colete os dados e outro que processe os dados. As razões para isso vão desde a simplificação do código até possibilitar o processamento de dados locais, não advindos da rede.

Isso pode ocorrer também em outras diversas situações, como um cálculo complicado de otimização de um processo, a execução de uma música MP3 ou tocar um vídeo DivX em tempo real, o que significa tocá-lo diretamente a partir do arquivo de dados comprimido: é desejável, por exemplo, que o processo que mostra o vídeo e o processo que o descomprime sejam independentes.

Essa separação tem várias vantagens:

a) Suporte a múltiplos formatos de dados: o processo que faz a descompressão fornece os dados prontos para serem apresentados pelo processo que mostra o vídeo. Desta forma, para suportar novos formatos de vídeo, basta trocar o processo que descomprime o dado. (poderia ser feito com um DLL em série, também, sem precisar do paralelismo)

b) Aproveitamento de diversas CPUs: em computadores multiprocessados, a forma de aproveitar o "poder de fogo" adicional é implementando processos paralelos, cada um fazendo parte do trabalho.

c) Simplificação da sincronia com o Hardware: um processo que receba dados da rede, precisa de uma disponibilidade muito grande para receber dados da placa de rede, coisa que teria problemas em fazer se estivesse no meio de um cálculo complexo e demorado. Da mesma forma, um software de vídeo tem que mostrar novos frames da imagem com um intervalo preciso (um determinado número de frames por segundo). Isso pode ser prejudicado se, no momento de apresentar a imagem, um frame estivesse sendo descomprimido.

A forma de conseguir estes benefícios através de múltiplos processos é pelo uso de *buffers*, ou seja, um local de troca de informações entre os processos. No caso do descompressor/player de vídeo, por exemplo, o descompressor pegaria os dados comprimidos e armazenaria no *buffer* os dados prontos para serem tocados, e o player pegaria estes dados do buffer e os apresentaria na tela. A figura 1 descreve o processo:

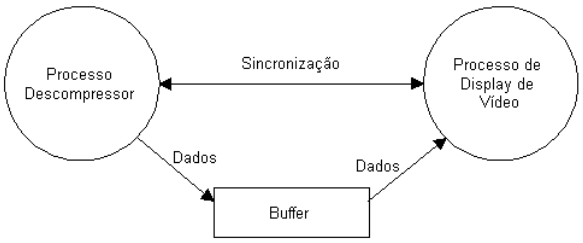


Figura 1: Processo de Comunicação entre dois Processos

Apesar da idéia simples, entretanto, o uso de vários processos para executar uma mesma tarefa implica em outras dificuldades, advindas do *compartilhamento de recursos* (o *buffer*, por exemplo), como a necessidade de sincronização entre os processos.

1. FORMAS COMUNS DE COMUNICAÇÃO ENTRE PROCESSOS

* Pipes (canos)
* Troca de Mensagens
* Memória Compartilhada

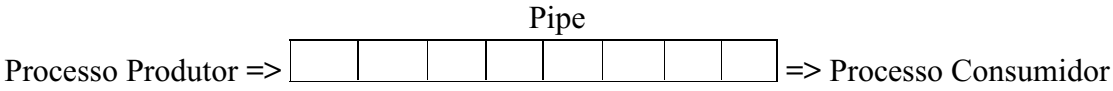
Existem três maneiras de comunicação entre processos: o uso de *pipes*, *a troca de mensagens* e a *memória compartilhada*.

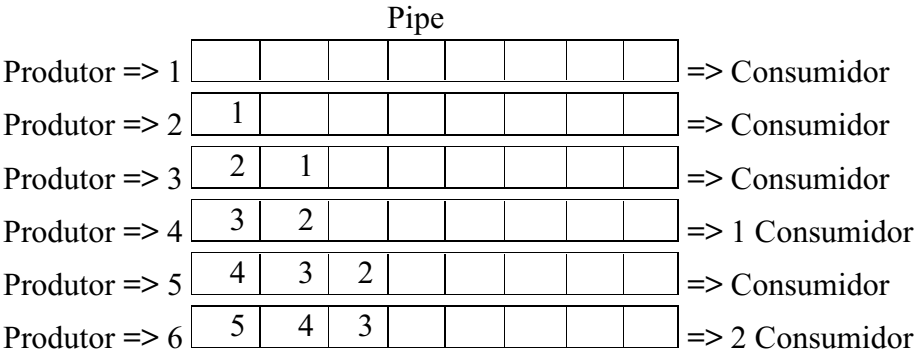
Cada uma destas formas de compartilhamento é voltada para um diferente uso, mas tem vantagens e desvantagens quanto às complexidades de gerenciamento.

1.1. *Pipes*

* Fila FIFO => ordem natural
* *Buffer*
* Produtor x Consumidor
* Envio / Recebimento com bloqueio
  - Controle pelo S.O.

Os pipes são estruturas fornecidas pelo sistema operacional de comunicação entre processos, que funcionam em um esquema de fila: o primeiro dado a ser colocado pelo processo que "enche" o *pipe* (cano) será o primeiro a ser retirado pelo processo que "esvazia" o *pipe*.





Como é possível ver, este sistema é interessante quando os dados são utilizados pelo consumidor na ordem em que chegam, ou seja, na ordem em que são colocados no *pipe* pelo produtor, como é comum ocorrer com dados do tipo "stream" (fluxo) como vídeo e áudio.

O problema que precisa ser gerenciado neste caso é que o processo consumidor pode ficar à espera de dados, caso o *buffer* (representado pelo *pipe*) esteja vazio **ou** o processo produtor ficar à espera de um espaço no *buffer* ser liberado, por estar cheio.

Em geral, entretanto, como se trata de uma estrutura totalmente operada pelo Sistema Operacional, não há maiores problemas por parte dos processos, a não ser a checagem que precisam executar, caso não seja desejável que o sistema operacional simplesmente "bloqueie" o processo enquanto a operação solicitada não puder ser atendida.

1.2. Troca de Mensagens

* SEND x RECEIVE
* Internamente: *pipes*
* Permite Sincronia: Leitura pressupõe Escrita
* Comunicação Direta: *Unnamed Pipe* (entre dois processos)
  x
* Comunicação Indireta: *Named Pipe* (vários processos)
  - "mailbox"
* Mensagens Síncronas (com espera) x Assíncronas (sem espera)

Uma forma de comunicação e sincronia normalmente fornecida pelos sistemas operacionais é a troca de mensagens, usando-se funções do tipo:

SEND (receptor, mensagem)
RECEIVE (transmissor, mensagem)

Embora o programador não precise tomar contato com esse detalhe, a troca de mensagens ocorre, em geral, através do uso de um ou mais *pipes* entre os processo. Apesar de ser, aparentemente, planejada para comunicação de conteúdo mais complexo, a troca de mensagens pode ser usada para sincronia entre processos, uma vez que uma mensagem só pode ser lida depois de ter sido enviada, o que possibilita o uso deste recurso para que um processo tenha de esperar uma mensagem de outro para poder continuar suas funções.

Dependendo da maneira como é implementado (através da rede, por exemplo), é possível que uma mensagem se perca. Neste caso é útil o uso de mensagem de recebimento (ACK) do receptor para o transmissor, de forma que o transmissor saiba se é necessário reenviar uma mensagem ou não.

A comunicação pode ser feita diretamente entre dois processos, com o *pipe/buffer* "sem nome", ou seja, ligando apenas os dois processos; uma outra forma de comunicação é a indireta, com um *pipe nomeado*, que funciona como uma caixa de correio (*mailbox*). Neste caso, os comandos SEND e RECEIVE serão ligeiramente diferentes:

SEND (mailbox, mensagem)
RECEIVE (mailbox, mensagem)

A vantagem, neste caso, é que vários processos podem se comunicar, interativamente, usando um único *buffer*. Um caso típico é quando existem diversos processos para executar uma determinada tarefa (descompressão de dados, por exemplo), em um computador com várias CPUs, e estes dados podem ter origem nas mais diversas aplicações.

Quando uma aplicação precisar que o dado seja descomprimido, ela envia para o mailbox adequado com o comando SEND. O dado ficará lá até que um dos processos

descompressor (o primeiro a estar disponível) pegue este dado e o processe, devolvendo o dado pronto por uma segunda mailbox.

1.2.1. Mensagens Síncronas e Mensagens Assíncronas

Independente da forma com que as mensagens são trocadas, elas podem ser síncronas ou assíncronas, ou seja, com ou sem espera.

Uma troca de mensagem é dita síncrona quando, após o SEND, o processo ficar "bloqueado" até que uma resposta seja recebida. Uma troca de mensagem é dita assíncrona quando, após o SEND, o processo continuar seu trabalho, recolhendo a mensagem de resposta posteriormente.

O mesmo conceito vale para o RECEIVE: se o processo que usa o receive ficar esperando a resposta, é uma troca de mensagem síncrona. Se o uso do RECEIVE não causar espera (caso não haja mensagens), a troca de mensagem é assíncrona.

1.3. Memória Compartilhada

* Memória Compartilhada
    - Sem ordem natural de acesso
    - *buffer* = estrutura de dados
        + atualizada pelo produtor
        + lida pelo consumidor
    - Uso pela leitura e escrita de valores nas variáveis
    - Cuidados!
        + Consistência: leituras x escritas simultâneas
* Seções Críticas (*Critical Sections*)
    - Bloqueio completo de outros processos
* Semáforos
    - Bloqueio Seletivo
    - Todo recurso deve ser solicitado previamente
    - Alocação de recurso: sinal muda para vermelho
    - Liberação de recurso: sinal muda para verde
    - E alocação com sinal vermelho?
        + Bloqueio do processo solicitante
* Semáforos e Seções Críticas => Exclusão Mútua

A memória compartilhada, ao contrário do *pipe*, é usada quando não existe uma ordem natural no "consumo" dos dados do *buffer*, isto é, quando os dados do buffer são uma estrutura de dados que está constantemente sendo atualizada pelo processo produtor, mas a consulta de seus dados pode ocorrer em diversas seqüências distintas.

Seu uso é como o de uma área de memória normal, onde se armazenam variáveis usuais. A diferença é que esta área "compartilhada" deve ter sido solicitada ao sistema operacional indicando esta necessidade e, então, o sistema criará uma área de memória em que dois ou mais processos podem escrever ou ler. Em um processo, as escritas e leituras desta área se dão pelo simples uso de uma variável, por exemplo.

Entretanto, há muitos cuidados que precisam ser tomados ao trabalhar com este tipo de compartilhamento. Por exemplo, se os dados compartilhados são os dados da conta de um cliente, com R$ 500,00 de saldo, e há dois processos em execução: um deles irá descontar R$ 10,00 do saldo, e o outro somar R$ 10,00. Neste caso, espera-se que o resultado final seja a manutenção dos R$ 500,00 na conta do cliente. Entretanto, isso pode não ocorrer de forma adequada, se alguns cuidados não forem tomados... e o resultado final ser R$ 490,00 ou mesmo R$ 510,00.

Para entender o porque disso, é preciso entender que os processos estão executando o seguinte código:

Processo de Crédito de R$10,00

a) Leitura do Saldo:	MOV EAX,[SALDO]
b) Somar 10,00 ao valor do saldo	ADD EAX,10
c) Escrever o novo valor na memória	MOV [SALDO],EAX

Processo de Débito de R$ 10,00

a) Leitura do Saldo	MOV EBX,[SALDO]
b) Subtrair 10,00 ao valor do saldo	SUB EBX,10
c) Escrever o novo valor na memória	MOV [SALDO],EBX

Considerando que as operações de escrita e leitura não podem ocorrer exatamente ao mesmo instante - pois a memória só permite um acesso por vez, mesmo em um sistema com múltiplas CPUs, é possível simular uma certa seqüência de execução da operação:

Instrução	Valor Saldo (mem)	Valor EAX (rg)	Valor EBX (rg)
MOV EAX,[SALDO]	500,00	500,00	???
MOV EBX,[SALDO]	500,00	500,00	500,00
ADD EAX,10	500,00	510,00	500,00
SUB EBX,10	500,00	510,00	490,00
MOV [SALDO],EAX	510,00	510,00	490,00
MOV [SALDO],EBX	490,00	510,00	490,00

Ora, como resolver este problema? Existem várias maneiras. Uma delas é usando uma **região crítica**, que é uma instrução de alto nível do sistema operacional que, momentaneamente, coloca em espera todos os outros processos que compartilham dados com esse, de todas as CPUs, até que se saia da região crítica. Assim, se fosse indicado:

Processo de Crédito de R$10,00

a) Entrada em Região Crítica	EnterCriticalSection()
b) Leitura do Saldo:	MOV EAX,[SALDO]
c) Somar 10,00 ao valor do saldo	ADD EAX,10
d) Escrever o novo valor na memória	MOV [SALDO],EAX
e) Sai da Região Crítica	ExitCritialSection()

Processo de Débito de R$ 10,00

a) Entrada em Região Crítica	EnterCritical Section()
b) Leitura do Saldo	MOV EBX,[SALDO]
c) Subtrair 10,00 ao valor do saldo	SUB EBX,10
d) Escrever o novo valor na memória	MOV [SALDO],EBX
e) Sai da Região Crítica	ExitCritialSection()

A execução ocorreria na seguinte ordem:

Instrução	Valor Saldo (mem)	Valor EAX (rg)	Valor EBX (rg)
EnterCriticalSection()	500,00	???	???
MOV EAX,[SALDO]	500,00	500,00	???
ADD EAX,10	500,00	510,00	???
MOV [SALDO],EAX	510,00	510,00	???
ExitCriticalSection()	510,00	510,00	???
EnterCriticalSection()	510,00	510,00	???
MOV EBX,[SALDO]	510,00	510,00	510,00
SUB EBX,10	510,00	510,00	500,00
MOV [SALDO],EBX	500,00	510,00	500,00
ExitCriticalSection()	500,00	510,00	500,00

Esta solução, entretanto, é um tanto quanto pobre e deselegante, já que bloqueia todos os processos que compartilham informações com este, **independente de o bloqueio ser necessário**. Assim, é necessário um sistema de "bloqueio seletivo", chamado **semáforo**.

O semáforo é fornecido, normalmente, pelo subsistema de gerenciamento e comunicação de processos, como um recurso para estes casos. Basicamente, todas os processos que usarem um recurso, assim que forem iniciar o uso deste recurso, devem solicitar o semáforo específico para aquele recurso.

Neste momento, o sistema operacional checa se o semáforo para este recurso está "verde", ou seja, não há nenhum outro processo utilizando aquele recurso. Se estiver "vermelho", o processo que acabou de requisitar o recurso será colocado em espera, até que o recurso seja liberado. Por outro lado, se o semáforo do recurso estiver "verde", o semáforo será alterado para "vermelho" e o recurso será normalmente acessado pelo processo.

Ao finalizar o uso do recurso, o processo deve liberar o recurso, ou seja, solicitar que o semáforo para aquele recurso volte a ficar "verde".

Ou seja, o mecanismo de uso é praticamente idêntico ao da seção crítica, com a diferença que, no semáforo, indicando um número para cada recurso que possa ser compartilhado, só serão bloqueados os processos que quiserem usar um recurso já sendo utilizado, permanecendo todos os outros em perfeita e contínua operação. Estes tipos de solução, do semáforo e da seção crítica, são chamados de ***métodos de exclusão mútua***.

2. DEADLOCK

- DeadLock => Processo esperando evento impossível
  - * Processo A espera B terminar, mas processo B espera A terminar
  - * Processo A tem recurso 1 e precisa do 2...
    - ...mas processo B tem recurso 2 e precisa do 1
- Prevenir, Detectar e Corrigir
- Prevenir => proteger toda a solicitação de recursos dentro de um semáforo
  - * Solução a partir do S.O.: Impossibilitar alocação de 2 recursos
- Detectar => Manter estrutura de dados e procurar ciclos de espera
- Corrigir => Fechar Processo x Liberar Recurso

DeadLock é uma situação em que um processo está esperando um evento que nunca acontecerá. Em geral o deadlock é originário de "exclusão mútua" no uso de recursos, anteriormente citada, em que um processo A espera o término do processo B, e o processo B espera pelo término do processo A, gerando uma situação do tipo "o que veio primeiro, o ovo ou a galinha?".

Uma situação mais comum, entretanto, é quando existem dois processos distintos que exigem dois recursos diferentes (ao mesmo tempo) para cumprir sua atividade. Dependendo da maneira com que os processos alocarem seus recursos, ocorrerá a situação de deadlock, como apresentado na figura 2.

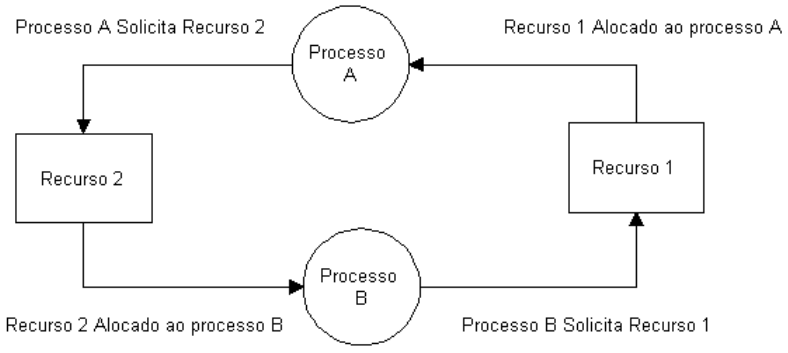


Figura 2: Processos em DeadLock

Na figura são representados dois processos, A e B, que precisam dos recursos 1 e 2 para operar. Ambos estão trabalhando absolutamente em paralelo e o processo A aloca primeiramente o recurso 1, para depois tentar alocar o recurso 2. No caso do processo B, ele aloca primeiramente recurso 2 e, depois, tenta alocar o recurso 1.

Entretanto, quando o processo A tenta alocar o recurso 2, ele já foi alocado para o processo B; por outro lado, quando o processo B tenta alocar o recurso 1, ele já foi alocado para o processo A. Ambos os processos serão colocados em espera até que um dos dois libere o recurso mas, como ambos foram colocados em espera, essa situação jamais ocorrerá.

Existem formas de prevenir, de detectar e de corrigir deadlocks. A prevenção pode se dar de diferentes formas, mas a mais comum é colocar todo o processo de solicitação de recursos um grupo de recursos protegido por um semáforo, de forma que se o processo A já estiver solicitado os recursos 1 e 2, o processo B será impedido, através do semáforo (que já estará "vermelho" até mesmo de começar sua solicitação. Mas esta é uma solução no software aplicativo, não tendo o Sistema Operacional responsabilidade sobre ela.

As maneiras para o sistema operacional evitar os deadlocks são bastante restritivas, como por exemplo impedir que um processo aloque dois recursos simultaneamente, ou seja, para alocar um segundo, precisa desalocar um primeiro.

Assim, alguns sistemas operacionais *detectam* e *corrigem* deadlocks, ao invés de preveni-los. Isso é feito mantendo uma tabela com as informações de todos os recursos alocados por cada processo, bem como de suas requisições no momento. Com estas informações, o sistema pode verificar se há uma espera circular como na figura 2, e assim detectar os processos que estejam em deadlock.

A correção do deadlock não é, exatamente, um processo simples. O método mais comum, embora existam outros, é a eliminação de um dos processos em deadlock, liberando assim seus recursos. Uma forma mais adequada seria liberar apenas os recursos de um dos processos, restabelecendo seu funcionamento posteriormente. Infelizmente isso normalmente não é possível.

3. PROGRAMAÇÃO PARALELA E DISTRIBUÍDA (OPCIONAL)

- Programação Direta
  - * Multi-Processos
  - * Multi-Threaded
- Programação Indireta: Linguagens Paralelas x Bibliotecas
  - * Cpar x Fortran (atual)
  - * MPI / OpenMP

Existem diversas formas de se desenvolver um aplicativo com processamento paralelo (várias CPUs no mesmo equipamento, usando a mesma memória), algumas delas úteis inclusive para o processamento distribuído (várias CPUs em equipamentos diferentes, com memórias diferentes para cada CPU).

A forma mais direta (e mais complexa) de se programar um aplicativo que tire proveito de processamento paralelo é através do uso de funções de criação de processos e de threads do sistema operacional. Esta solução exige que a maior parte dos controles de sincronia seja feito pelo usuário, com o uso de mensagens e/ou semáforos.

Há, porém, formas indiretas e mais simples, como o uso de linguagens preparadas para o processamento paralelo, como o CPar (C Paralelo, desenvolvido na USP) ou as versões mais recentes do Fortran. Estas linguagens possuem maneiras de se definir funções que devem ser executadas em paralelo, pontos de sincronia entre processos/thread, além de instruções que fazem paralelismo automático, como a instrução "forall". Algumas destas linguagens são capazes de lidar com processamento distribuído também.

Para as linguagens mais usuais, que não possuem instruções de processamento paralelo e distribuído, existem bibliotecas que, fazendo uso das funções existentes no sistema operacional, encapsulam grande parte das complexidades da programação paralela ou distribuída direta, fornecendo ferramentas simples para trocar mensagens síncronas ou não, realizar sincronia etc. Exemplos deste tipo de biblioteca são o MPI e o OpenMP.

4. ATIVIDADE

Sua equipe foi contratada para especificar alguns critérios de um sistema de suporte de atendimento ao cliente (SAC) utilizado por uma nova empresa, a NetShop.

O objetivo da NetShop é permitir que o sistema de suporte seja suficiente para troca de mensagens de texto entre os usuários e também a troca de arquivos, em um formato parecido com o do MSN.

Adicionalmente, foi definido que o sistema deve ser executado com três processos: um responsável pela interface com o usuário (processo principal), um responsável pelo envio de informações e outro pelo recebimento de informações. Ocorre que, por razões de economia de recursos do servidor, foi definido que a mesma conexão de rede deve ser usada para o envio e o recebimento de dados. Em outras palavras, não é possível enviar e receber ao mesmo tempo. Entretanto, em momento alguma a interface da aplicação (ou do sistema) deve ficar bloqueada.

Especifique os mecanismos para troca de mensagens texto e arquivos entre o processo principal e os processos de envio/recebimento, além de descrever o mecanismo necessário para garantir que os processos de envio e recebimento de mensagens nunca estarão utilizando a rede ao mesmo tempo.

5. BIBLIOGRAFIA

MACHADO, F. B; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª. Ed. LTC, 2007.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª.Ed. Prentice Hall, 2003.

SILBERSCHATZ, A; GALVIN, P. B. **Sistemas operacionais**: conceitos. PrenticeHall, 2000.

DAVIS, W.S. **Sistemas Operacionais**: uma visão sistêmica. São Paulo: Ed. Campus, 1991.