

Sistemas Operacionais

Erico Meneses Leão

APRESENTAÇÃO

Sistemas Operacionais (SO) é uma disciplina essencial e obrigatória em praticamente todos os cursos universitários de Computação. De fato, um Sistema Operacional é uma parte essencial de qualquer Sistema de Computação. Trata-se, então, de uma área de pesquisa bastante interessante, que evolui a todo o momento, posto que os Sistemas de Computação estão em constante evolução.

Um Sistema Operacional é um conjunto de rotinas executado pelo processador (CPU), de forma análoga aos programas de usuários. A principal função dessas rotinas é controlar o funcionamento do Sistema Computacional, gerenciando de forma otimizada os recursos disponíveis, como processadores, memória e dispositivos de entrada e saída, além de, na medida do possível, fazer a interface entre o *hardware* e o usuário final, procurando esconder sua complexidade e mostrando um ambiente agradável e de fácil utilização.

Se não existisse o Sistema Operacional, o usuário, para poder manipular o computador, deveria conhecer os diversos detalhes de *hardware*, o que tornaria o seu trabalho mais cansativo, lento e impróprio, com uma grande perspectiva de erros.

O objetivo desta apostila é proporcionar ao leitor um bom entendimento sobre o fantástico mundo dos Sistemas Operacionais. Ao longo dos capítulos iremos abordar os conceitos básicos que envolvem toda a área, além de detalhar cada estrutura formadora de um Sistema Operacional. Cada capítulo é acompanhado de embasamento teórico sobre cada parte de um Sistema Operacional,

além de exercícios para praticar o assunto. A bibliografia e a webliografia ao fim das notas são mais do que suficiente para que o leitor se aprofunde na teoria apresentada em cada unidade.

Esta apostila é completamente baseada em notas de aula do autor da disciplina de Sistemas Operacionais, ministrada no curso de Bacharelado em Ciência da Computação da Universidade Federal do Piauí, além de livros clássicos de Sistemas Operacionais, como Sistemas Operacionais Modernos, de *Andrew Tanenbaum* (referência básica e principal), Fundamentos de Sistemas Operacionais, de *Abraham Silberschatz* e *Operating Systems (J. Bacon)*. Outro livro que foi levado em consideração como base dessa apostila é o de Sistemas Operacionais do professor Rômulo Oliveira da Universidade Federal de Santa Catarina, livro este bastante didático e utilizado como referência na maioria dos cursos de Sistemas Operacionais no país.

Conteúdo desta Apostila

Na **Unidade I** apresentaremos o conceito de Sistemas Operacionais, o que eles fazem e como são projetados. Abordaremos o conceito de um Sistema Operacional partindo da visão de uma máquina estendida, que procura esconder a complexidade de *hardware* para o usuário e da visão de um gerenciador de recursos, que procura gerenciar todos os recursos disponíveis pelo Sistema Computacional. Nesta unidade, mostraremos também um histórico dos Sistemas Operacionais, mostrando sua evolução durante os anos, sempre acompanhado da evolução dos Sistemas Computacionais. Além disso, trataremos de classificar os Sistemas Operacionais e apresentar como um Sistema Operacional é estruturado. Ao fim de cada unidade teremos exercícios a fim de que o aluno possa praticar os conhecimentos adquiridos.

Na **Unidade II** começaremos a descrever a primeira parte de um projeto de Sistemas Operacionais que trata do gerenciamento de processos. Nesta unidade mostraremos como é projeto o modelo de processos, abordando os conceitos básicos, comunicação inter-processos (CIP), métodos de exclusão mútua, problemas clássicos de CIP e o gerenciamento do processador, abordando os algoritmos clássicos de escalonamento de processos, que a partir de uma lista de processos escolhe qual utilizar a CPU para executar suas atividades.

Na **Unidade III** está destinada ao gerenciamento dos dispositivos de entrada e saída. Através dos dispositivos de entrada e saída é feito a ligação entre o sistema e o mundo exterior. Na unidade mostraremos os princípios de *hardware* e de *software* dos dispositivos de entrada e saída. Comumente a quantidade desses dispositivos é muito menor que a quantidade de processos que necessitam deles. Dessa forma, o Sistema Operacional deve gerenciá-los a fim de se evitar problemas. Veremos também nesta unidade as situações de impasses (*deadlocks*) entre os processos.

Na **Unidade IV**, por sua vez, trataremos do gerenciamento de um recurso altamente importante para qualquer sistema: a memória. O ideal seria que o usuário tivesse uma quantidade infinita de memória, rápida, não-volátil e com um preço acessível. Porém, essas características são contraditórias e, assim, o sistema deve gerenciar a memória disponível no Sistema Computacional que, comumente, é limitada. Veremos também nesta unidade o método de gerenciamento conhecido como memória virtual.

Por fim, na **Unidade V** mostraremos a parte do Sistema Operacional mais visível pelo usuário: o Sistema de Arquivo. As informações necessitam ser gravadas de forma permanente e isso acontece através da abstração arquivos. Nesta unidade

mostraremos as características gerais de arquivos e diretórios e como eles podem ser implementados.

Boa Leitura!!

Erico Meneses Leão

SUMÁRIO GERAL

UNIDADE I – VISÃO GERAL

1. INTRODUÇÃO

1.1. Definição de Sistemas Operacionais

1.2. Objetivos de um Sistema Operacional

1.3. História dos Sistemas Operacionais

1.3.1. Início

1.3.2. Primeira Geração (1945-1955)

1.3.3. Segunda Geração (1956-1965)

1.3.4. Terceira Geração (1966-1980)

1.3.5. Quarta Geração (1981-Atual)

1.4. Classificação dos Sistemas Operacionais

1.4.1. Sistemas Monoprogramáveis ou Monotarefas

1.4.2. Sistemas Multiprogramáveis ou Multitarefas

1.4.3. Sistemas Multiprocessadores

1.5. Estrutura do Sistema Operacional

1.5.1. Chamadas de Sistemas

WEBLIOGRAFIA

REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE II – GERENCIAMENTO DE PROCESSOS

1. INTRODUÇÃO AO MODELO DE PROCESSOS

1.1. Conceito de Processos

1.2. Criação de Processos

1.3. Estados de um Processo

1.4. Tabelas de Processos

1.5. *Threads*

- 2. COMUNICAÇÃO INTERPROCESSO
 - 2.1. Condições de Corrida
 - 2.2. Seções Críticas
 - 2.3. Exclusão Mútua com Espera Ativa
 - 2.3.1. Desativando Interrupções
 - 2.3.2. Variáveis de Bloqueio
 - 2.3.3. Alternância Estrita
 - 2.3.4. A Solução de Peterson
 - 2.4. Problema dos Produtores e Consumidores
 - 2.5. Semáforos
- 3. PROBLEMAS CLÁSSICOS DE CIP
 - 3.1. Problema do Jantar dos Filósofos
 - 3.2. Problema do Barbeiro Adormecido
- 4. ESCALONAMENTO DE PROCESSOS
 - 4.1. Algoritmo de Escalonamento FIFO (*First in First out*)
 - 4.2. Algoritmo de Escalonamento Menor Tarefa Primeiro
 - 4.3. Algoritmo de Escalonamento *Round Robin*
 - 4.4. Algoritmo de Escalonamento por Prioridades
 - 4.5. Algoritmo de Escalonamento Múltiplas Filas
- WEBLIOGRAFIA
- REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE III – GERENCIAMENTO DE ENTRADA E SAÍDA

- 1. INTRODUÇÃO
- 2. PRINCÍPIOS DE *HARDWARE* DE E/S
 - 2.1. Dispositivos de Entrada e Saída
 - 2.2. Controladoras de Dispositivos
 - 2.2.1. Controladoras que suportam Acesso Direto à Memória
- 3. PRINCÍPIOS DE *SOFTWARE* DE E/S
 - 3.1. Manipuladores de Interrupções
 - 3.2. *Drivers* de Dispositivos

3.3. Software de E/S Independente de Dispositivo

3.4. Software de E/S de Nível de Usuário

4. DEADLOCKS

4.1. Condições para um Impasse

4.2. Modelagem de Impasses através de Grafos

4.3. Métodos de Lidar com *Deadlocks*

4.3.1. Algoritmo do Avestruz

4.3.2. Detecção e Recuperação

4.3.3. Prevenção de Impasses

4.3.4. Impedimento de Impasses

WEBLIOGRAFIA

REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE IV – GERENCIAMENTO DE MEMÓRIA

1. INTRODUÇÃO

2. GERENCIAMENTO BÁSICO DE MEMÓRIA

3. GERENCIA DE MEMÓRIA PARA MULTIPROGRAMAÇÃO

3.1. Alocação com Partições Variáveis

3.1.1. Gerenciamento de Memória com Mapa de Bits

3.1.2. Gerenciamento de Memória com Listas Encadeadas

4. MEMÓRIA VIRTUAL

4.1. Paginação

4.2. TLB – *Translation Lookside Buffers*

5. ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINAS

5.1. Algoritmo de Substituição de Página Ótimo

5.2. Algoritmo de Substituição de Página Não Recentemente Utilizada

5.3. Algoritmo de Substituição de Página FIFO

5.4. Algoritmo de Substituição de Página de Segunda Chance

5.5. Algoritmo de Substituição de Página do Relógio

5.6. Algoritmo de Substituição de Página menos

Recentemente Utilizada
WEBLIOGRAFIA
REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE V – SISTEMAS DE ARQUIVOS

1. INTRODUÇÃO AOS SISTEMAS DE ARQUIVOS
 - 1.1. Arquivos
 - 1.1.1. Nomeação de Arquivos
 - 1.1.2. Estruturas de Arquivos
 - 1.1.3. Tipos de Arquivos
 - 1.1.4. Acesso aos Arquivos
 - 1.1.5. Atributos de Arquivos
 - 1.1.6. Operações com Arquivos
 - 1.2. Implementação de Arquivos
 - 1.2.1. Alocação Contígua
 - 1.2.2. Alocação por Lista Encadeada
 - 1.2.3. Alocação por Lista Encadeada Usando Índice
 - 1.3. Diretórios
 - 1.3.1. Organização de Sistemas de Diretórios
 - 1.3.2. Nomes de Caminhos
 - 1.3.3. Operações com Diretórios
 - 1.4. Implementação de Diretórios
- WEBLIOGRAFIA
- REFERÊNCIAS BIBLIOGRÁFICAS



Para iniciarmos nosso curso de Sistemas Operacionais (SO) devemos compreender as definições básicas que envolvem a área. A partir desse momento, a fim de evitar muitas repetições, iremos tratar Sistemas Operacionais pelo termo SO. Nossa primeira unidade é dedicada à visão geral sobre SO. Nela trataremos de definir o que é um Sistema Operacional, tomando por base duas visões: a visão de SO como uma máquina estendida e o SO como gerenciador dos recursos (*Andrew Tanenbaum*).

A definição de Sistemas Operacionais do ponto de vista de uma máquina estendida procura esconder a complexidade do *hardware* computacional. Já a definição de Sistemas Operacionais do ponto de vista gerenciador de recursos trata de organizar os recursos computacionais disponíveis, evitando assim possíveis problemas, como inconsistências e disputas entre os programas.

Continuando na unidade, traremos um breve histórico dos Sistemas Operacionais, que vem junto da evolução dos Sistemas Computacionais, além da classificação dos SO's.

Ao fim da unidade, o leitor deverá ter a capacidade de entender como e por quê surgiram os Sistemas Operacionais, além de conhecer os tipos de Sistemas Operacionais. Ao fim da Unidade,

o leitor terá disponíveis exercícios e bibliografia clássica, o qual ele poderá futuramente se aprofundar no estudo de Sistemas Operacionais, e quem sabe, ingressar nesta grande área de estudo e pesquisas.

SUMÁRIO

UNIDADE I – VISÃO GERAL

1. INTRODUÇÃO

- 1.1. Definição de Sistemas Operacionais
- 1.2. Objetivos de um Sistema Operacional
- 1.3. História dos Sistemas Operacionais
 - 1.3.1. Início
 - 1.3.2. Primeira Geração (1945-1955)
 - 1.3.3. Segunda Geração (1956-1965)
 - 1.3.4. Terceira Geração (1966-1980)
 - 1.3.5. Quarta Geração (1981-Atual)
- 1.4. Classificação dos Sistemas Operacionais
 - 1.4.1. Sistemas Monoprogramáveis ou Monotarefas
 - 1.4.2. Sistemas Multiprogramáveis ou Multitarefas
 - 1.4.3. Sistemas Multiprocessadores
- 1.5. Estrutura do Sistema Operacional
 - 1.5.1. Chamadas de Sistemas

WEBLIOGRAFIA

REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE I

VISÃO GERAL

1. INTRODUÇÃO

Antes de começarmos a detalhar os principais componentes formadores de um Sistema Operacional (SO), devemos primeiramente entender alguns conceitos e funcionalidades básicas. Devemos compreender que, se não existisse o Sistema Operacional, até seria possível utilizar os Sistemas Computacionais disponíveis atualmente, porém, a complexidade e dificuldade na sua manipulação os tornariam bem pouco atraentes.

Se não existisse o Sistema Operacional, o usuário, para poder manipular o computador ou qualquer Sistema Computacional, deveria conhecer os diversos detalhes de *hardware* (tarefa árdua), o que tornaria o seu trabalho mais cansativo, lento e impróprio, sem considerar a grande quantidade de problemas e erros nos resultados obtidos.

Como explicitado por Rômulo (2008) em seu livro de Sistemas Operacionais, “*em torno de um computador, existem usuários com problemas a serem resolvidos*”, problemas esses que podem se estender de simples entretenimento, até edição de textos, figuras e atividades mais complexas, como uma análise estatística ou um gerenciamento de uma empresa por completo. O *software*, de um modo geral, é utilizado para solucionar os problemas do usuário, enquanto que o *hardware* do computador é o dispositivo físico capaz de executar esses *softwares*. Esses *softwares*, responsáveis por realizar as atividades dos usuários, comumente são chamados de programas aplicativos.

Com base no que já foi dito, surge a seguinte pergunta: mas onde entra o Sistema Operacional? Assim, o Sistema Operacional

pode ser caracterizado como uma camada de *software* localizado entre o *hardware* e os programas aplicativos que executam as atividades dos usuários, como esboçado na Figura 1.

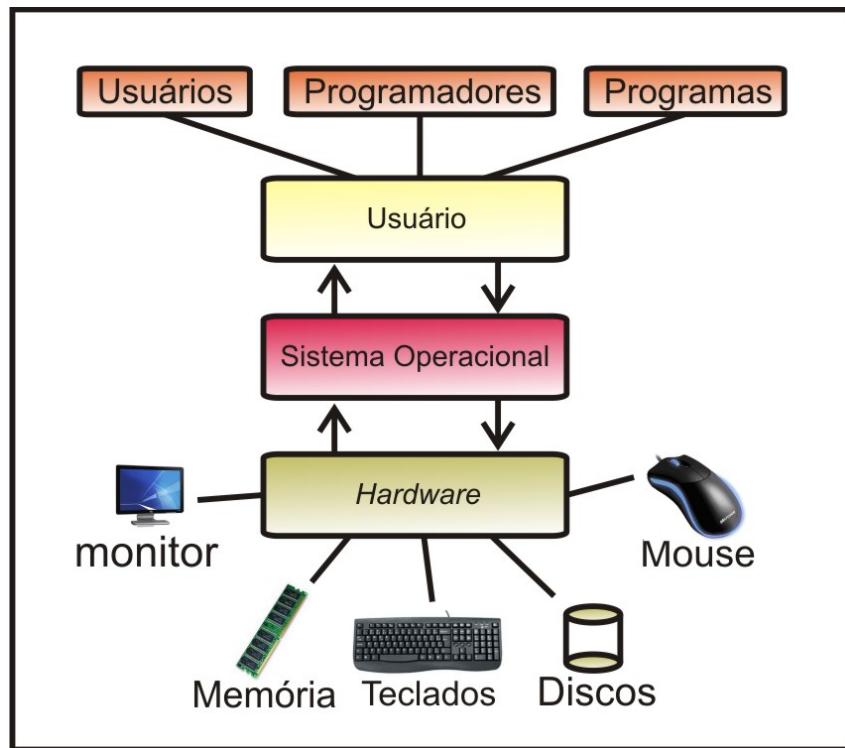


Figura 1: Sistema Computacional composto pelo hardware, programas de sistemas e programa aplicativos.

Como pode ser visto na Figura 1, levando em consideração que temos uma camada intermediária, o SO entre os programas aplicativos e o *hardware* em si, o usuário não necessita conhecer toda a complexidade de implementação do *hardware* do Sistema Computacional para, assim, poder utilizá-lo. O Sistema Operacional, de fato, opera como uma interface entre o usuário e o dispositivo físico em si, no qual o usuário, quando necessita acessá-lo, faz essa solicitação diretamente ao Sistema Operacional.

1.1. Definição de Sistemas Operacionais

Segundo *Tanenbaum*, podemos definir um Sistema Operacional levando em consideração dois pontos de vistas:

- O Sistema Operacional como uma Máquina estendida;
- O Sistema Operacional como gerenciador de recursos.

De fato, unindo e entendendo os dois pontos de vista, o aluno terá total condição de definir um bom conceito de Sistemas Operacionais. Examinaremos com detalhes esses dois pontos de vista.

O Sistema Operacional como uma Máquina estendida

O usuário (que pode ser um programador ou um usuário final), comumente, não está interessado em saber os detalhes funcionais dos dispositivos. Como exemplo, o usuário não quer saber o que é preciso, a nível de *hardware*, para que seja lido uma determinada informação de um disquete ou de um disco rígido (tarefa bem complexa, que exige o conhecimento de registradores, motores, cilindros e outros dispositivos físicos). O usuário deseja ter uma interface mais palpável e mais simples de lidar. No caso dos discos, por exemplo, uma abstração típica seria que o disco contenha um conjunto de nomes de arquivos. A partir desses nomes, é possível realizar as operações básicas (abrir, ler, escrever e fechar), sem se importar qual a velocidade e estado atual do motor, por exemplo.

Assim, o Sistema Operacional aparece como o programa que esconde do usuário a complexidade do *hardware* e apresenta uma visão fácil e simples para as operações sobre os dispositivos. Essa visão é equivalente a uma máquina estendida ou máquina virtual, mais fácil de lidar.

O Sistema Operacional como gerenciador de recursos

Por outro lado, o Sistema Computacional é composto de uma série de recursos, no qual podemos enumerar: processadores,

memórias, discos, mouses, teclados, impressoras, placas de rede e uma infinidade de dispositivos em geral. Dessa forma, o Sistema Operacional aparece como sendo o responsável por organizar e alocar de forma ordenada todos esses recursos disponíveis.

Essa tarefa, em uma primeira vista, pode parecer simples. Porém, quando se tem vários programas disputando os recursos, que são limitados, é necessário utilizar técnicas de alocação dos dispositivos, a fim de se evitar inconsistências e, até mesmo, situações que resultem numa parada do sistema de uma forma geral.

1.2. Objetivos de um Sistema Operacional

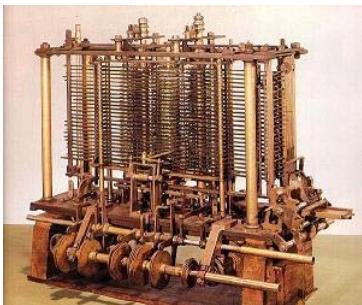
Para o desenvolvimento de um Sistema Operacional devemos então ter como metas atingir os seguintes objetivos:

- Tornar a utilização do computador eficiente e conveniente, a fim de ter um ganho de produtividade e, dessa forma, utilizar o Sistema Computacional para agilizar as atividades do dia-a-dia;
- Garantir a integridade e segurança dos dados armazenados e processados pelos programas e dos recursos físicos disponíveis.

1.3. História dos Sistemas Operacionais

Os Sistemas Operacionais, ao longo dos anos, vêm se desenvolvendo e ganhando novas características, sendo necessário partirmos ao seu histórico para que possamos compreender como se deu essa evolução. Partindo do pressuposto que a história dos Sistemas Operacionais sempre esteve intimamente vinculado à história das arquiteturas de computadores, iremos fazer um breve resumo dos principais eventos relacionados à evolução dos Sistemas Operacionais.

1.3.1. Início



Máquina Analítica de Babbage.

O primeiro computador digital, de fato, foi projetado por volta da década de 1820 pelo matemático Charles Babbage e intitulada como motor analítico. Esta máquina, por se tratar de um equipamento puramente mecânico e a tecnologia da época não permitir a construção de engrenagens de alta precisão o qual Babbage necessitava, nunca funcionou adequadamente. Assim, o motor analítico de Babbage não possuía Sistema Operacional.

Um dado interessante está no fato de que Babbage sabia que era preciso de um *software* para o seu motor analítico e, assim, contratou uma jovem chamada Ada Lovelace como primeiro programador do mundo. O nome da linguagem de programação Ada foi criado em homenagem a esta jovem.

1.3.2. Primeira Geração (1945-1955)



Computador de Primeira Geração.

Impulsionado pela Segunda Guerra Mundial, surgiram os grandes computadores digitais, formados por milhares de válvulas e que ocupavam salas inteiras. Estes computadores, desenvolvidos por *Howard Aiken* e *John von Neumann*, eram extremamente lentos.

Para trabalhar nesta máquina era necessário o conhecimento do funcionamento do seu *hardware*, onde a programação era feita através de linguagem de máquina, frequentemente ligando painéis de conectores com fios para o controle das funções básicas. Nessa época, ainda não existia o conceito de Sistema Operacional. Por esse fato, esta geração ficou conhecida como a geração das válvulas e painéis de conectores.



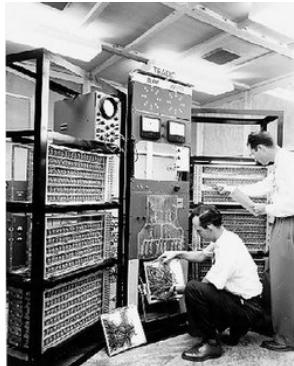
Transistor

1.3.3. Segunda Geração (1956-1965)

Com o desenvolvimento dos transistores, os computadores sofreram um enorme avanço, tornando-se mais confiáveis a fim de serem comercializados.

Nesta época, com o surgimento das primeiras linguagens de programação, os programas deixaram de ser feitos diretamente no

hardware, facilitando assim o processo de desenvolvimento de programas.



Computador de Segunda Geração

As máquinas de transistores eram armazenadas em salas e operadas por equipes especiais. Para executar uma atividade, o programador escrevia o seu programa, inicialmente, em um papel (em FORTRAN ou ASSEMBLY), e transformava este em cartões perfurados. O seu conjunto de cartões era levado para a sala onde se encontrava a máquina e era entregue diretamente aos operadores.

Os cartões eram carregados em fitas magnéticas, que eram lidas pelo computador, que executava um programa de cada vez, gravando o resultado do processamento em uma fita de saída. Esse tipo de processamento, onde um lote de programas era submetido ao computador, deu-se o nome de processamento em lotes ou processamento em *batch*. A Figura 2 visualiza este procedimento.

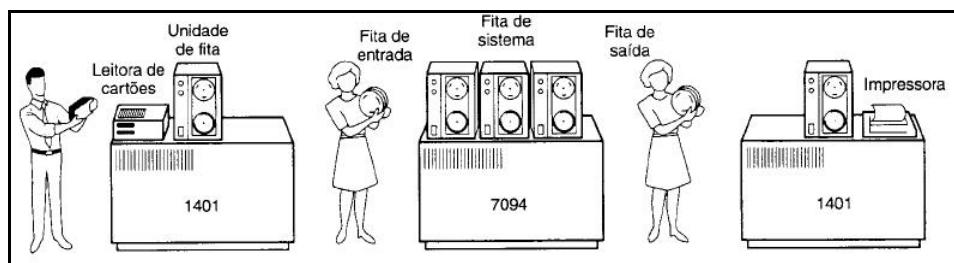


Figura 2: Sistema de processamento em lote.

1.3.4. Terceira Geração (1966-1980)

A terceira geração é conhecida com geração dos circuitos integrados (Cl's) e da multiprogramação, diminuindo consideravelmente o preço do computador, possibilitando assim sua aquisição por empresas. Esta época se caracteriza pelo grande aumento do poder de processamento e, também, a diminuição dos equipamentos.

Nesta época, a IBM lançou o System/360, que era uma série de computadores pequena, poderosa e, sobre tudo, compatível. O 360 foi projetado para manipular cálculos tanto científicos como

comerciais, ou seja, em uma única família de máquinas era possível satisfazer as necessidades de praticamente todos os clientes.

Porém, para atender todas as aplicações e periféricos disponíveis por essa família de máquinas, a IBM teve que desenvolver um Sistema Operacional (OS/360) extremamente grande e complexo, posto que as aplicações disponíveis, comumente, eram contraditórias. Este Sistema Operacional consistia de milhões de linhas de linguagem *assembler* escrita por milhares de programadores e muitos *bugs*, que exigiam versões e mais versões a fim de corrigi-los.

Apesar de todos os problemas, o OS/360 e os Sistemas Operacionais semelhantes atenderam a maioria dos seus clientes razoavelmente bem. Além disso, eles lançaram várias técnicas utilizadas até hoje, como exemplo a **multiprogramação**. A multiprogramação consistia em dividir a memória em várias partições a fim de permitir que várias tarefas sejam carregadas em cada partição. Enquanto uma tarefa esperava alguma operação de Entrada ou Saída, outra tarefa poderia usar o processador (CPU).

Outro recurso disponível nos Sistemas Operacionais da terceira geração era a capacidade de ler *jobs* (tarefas) de cartões para o disco. Assim, sempre que um *job* acabava sua execução, o Sistema Operacional podia carregar um novo *job* do disco para a partição e executá-lo. Esta técnica é conhecida como *spooling*.

Entretanto, os Sistemas Operacionais ainda eram basicamente sistemas em lote e que não exigiam comunicação com o usuário. Assim, muitos programadores sentiam falta das máquinas de primeira geração, que eram disponibilizadas por completa para eles e, assim, podiam depurar seus programas.

Assim, a multiprogramação evoluiu preocupada em oferecer aos usuários tempos de respostas razoáveis e uma interface cada vez mais amigável. Para tal, cada programa na memória utilizaria o processador em pequenos intervalos de tempo. Esse sistema de divisão de tempo ficou conhecido como compartilhamento de Tempo (*time-sharing*).

A terceira geração também é marcada pelo surgimento do Sistema Operacional UNIX, escrito em linguagem de programação de alto nível, que se tornou popular no mundo acadêmico, entre órgãos do governo e entre muitas empresas.

1.3.5. Quarta Geração (1981-Atual)

De fato, a década de 1980 é caracterizada pelo surgimento dos computadores pessoais. Os computadores pessoais se tornaram possíveis devido ao advento de novas tecnologias, impulsionados pelo avanço da indústria de *hardware*, com a introdução de novos circuitos integrados. Os computadores pessoais permitiram que as pessoas pudessem ter seu próprio computador.

Os equipamentos desta geração se tornaram cada vez menores, mais velozes e, principalmente, mais baratos. Esses novos equipamentos, com alta disponibilidade de poder de computação, especialmente a computação altamente interativa, normalmente com excelentes gráficos, levaram ao crescimento de uma importante indústria, a indústria de *softwares* para computadores pessoais.

Dois Sistemas Operacionais inicialmente dominaram o cenário dos computadores pessoais: o MS-DOS (*Microsoft*) e o UNIX. O MS-DOS foi amplamente utilizado no IBM PC e em computadores com a tecnologia Intel. Esse Sistema Operacional evolui para o sistema conhecido como *Windows*.

Outra evolução que surgiu nesta geração foi o crescimento de redes de computadores pessoais executando Sistemas Operacionais de rede e Sistemas Operacionais distribuídos. Em um Sistema Operacional de rede, os usuários podem conectar-se a máquinas remotas e copiar arquivos de uma máquina para a outra. Em um Sistema Operacional distribuído os programas dos usuários podem ser executados em vários computadores, porém, estes vêem o sistema como único.

Alguns autores ainda apontam uma quinta geração, que engloba o desenvolvimento cada vez maior da indústria do *hardware* e do *software*, além do desenvolvimento das telecomunicações,



permitindo o funcionamento de sistemas e aplicações distribuídas, que figuram nossa realidade.

1.4. Classificação dos Sistemas Operacionais

Os Sistemas Operacionais evoluíram juntamente com a evolução do *hardware* e das aplicações por ele suportada. Muitos termos inicialmente introduzidos para definir conceitos e técnicas foram substituídos por outros.

Abordaremos neste tópico, os diversos tipos de Sistemas Operacionais classificados quanto ao seu tipo de processamento (Figura 3), apontando suas principais características.

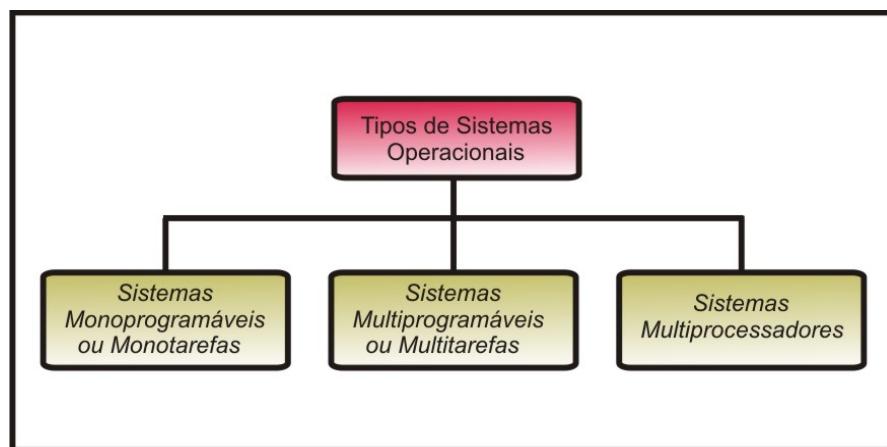


Figura 3: Tipos de Sistemas Operacionais.

1.4.1. Sistemas Monoprogramáveis ou Monotarefas

Os Sistemas monoprogramáveis ou monotarefas são caracterizados por alocar o Sistema Computacional disponível exclusivamente para um único programa, ou seja, um programa tem todos os dispositivos, incluindo periféricos, memória e processador disponível durante todo o tempo em que ele está ativo, mesmo se não estiver usando.

Nesse caso, se o programa estivesse executando uma operação de entrada e saída, o processador continuaria disponível e

alocado a ele, mesmo ele não estando utilizando (o processador ficaria ocioso).

Os primeiros sistemas operacionais eram tipicamente voltados para a execução de um único programa. Os sistemas monoprogramáveis estão tipicamente relacionados ao surgimento dos primeiros computadores na década de 1960 e se caracterizam por permitir que todos os recursos do sistema fiquem exclusivamente dedicados a uma única tarefa, como ilustrado na Figura 4.

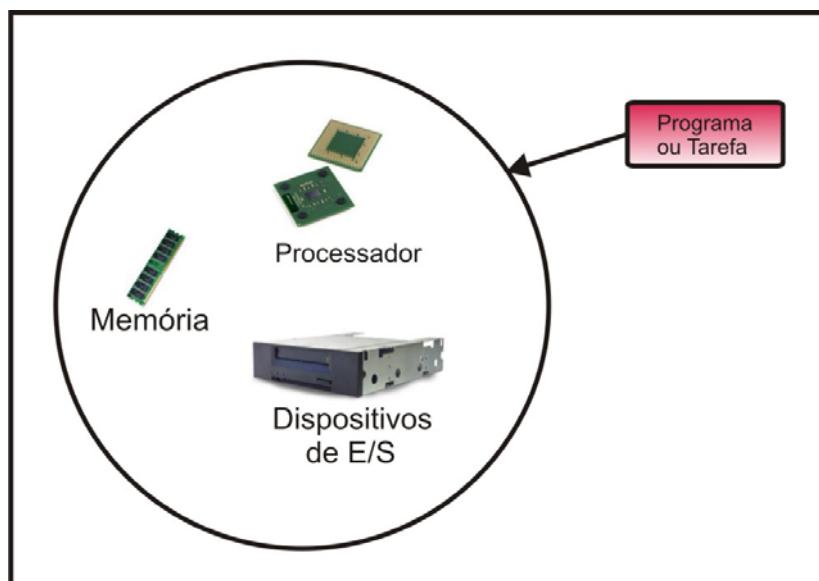


Figura 4: Sistemas Monoprogramáveis.

Os Sistemas monoprogramáveis são de simples implementação, comparado com sistemas multiprogramáveis e multiprocessadores, posto que não necessite muita preocupação com compartilhamento e concorrência.

1.4.2. Sistemas Multiprogramáveis ou Multitarefas

Os Sistemas multiprogramáveis já são bem mais complexos que os Sistemas monoprogramáveis. Neste tipo de sistema, os diversos recursos do Sistema Computacional são divididos pelas várias tarefas ou programas que necessitam utilizá-los, como visualizado na Figura 5.

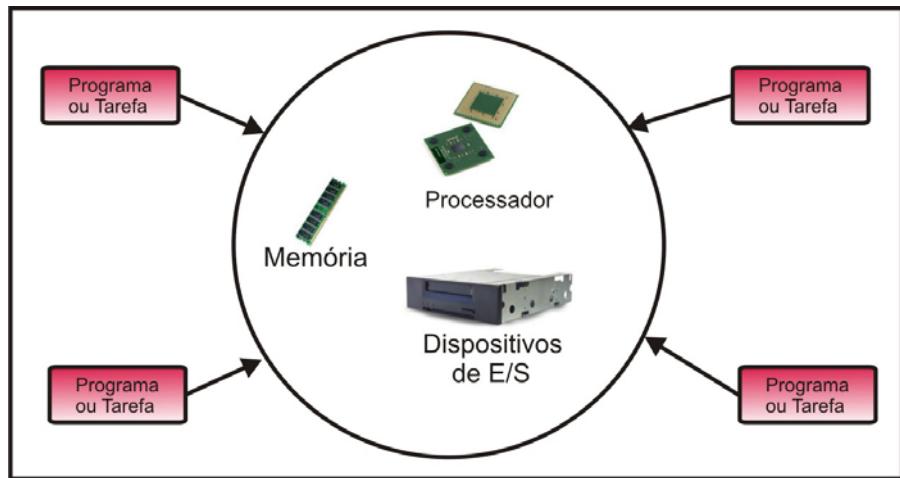


Figura 5: Sistemas Multiprogramáveis.

A grande vantagem dos Sistemas multiprogramáveis está no fato dos recursos poderem ser divididos entre os vários programas, ganhando tempo e, naturalmente, aumentando a produtividade do usuário. Assim, por exemplo, em um caso onde um programa necessite fazer uma operação de E/S e não esteja utilizando a CPU, esta poderá ser disponibilizada para outro programa.

Os Sistemas multiprogramáveis podem ser classificados pelo número de usuários que interagem com o sistema e pela forma com que suas aplicações são gerenciadas.

Os Sistemas multiprogramáveis, quanto ao número de usuários que interagem com o Sistema, podem ser classificados a seguir:

- Sistemas monousuários: sistemas multiprogramáveis onde apenas um usuário interage com o sistema, podendo realizar várias atividades ao mesmo tempo, como edição de texto, impressão e acesso a Internet, por exemplo.
- Sistemas multiusuários: sistemas multiprogramáveis onde vários usuários podem estar conectados e interagindo com o sistema.

Os Sistemas multiprogramáveis, quanto à forma que suas aplicações são gerenciadas, podem ser classificados como: sistemas *batch*, de tempo compartilhado ou de tempo real.

Sistemas *Batch*

Os sistemas batch foram os primeiros Sistemas Operacionais multiprogramáveis. Este tipo de sistema não necessitava da interação do usuário, o qual o programa era carregado no computador a partir de algum tipo de memória secundária e eram executados sequencialmente.

Sistemas de Tempo Compartilhado

Os Sistemas de tempo compartilhado (*time-sharing*) permitem que diversos programas sejam executados a partir da divisão do tempo do processador. Cada programa utiliza o processador por uma fatia de tempo e pode ser interrompido caso o seu tempo termine. O sistema cria um ambiente de trabalho próprio, dando a impressão para o usuário que o sistema está todo disponível a ele.

Sistemas de Tempo Real

Os sistemas de tempo real têm características semelhantes aos sistemas de tempo compartilhado, porém, suas aplicações possuem requisitos temporais, ou seja, possuem um tempo máximo para serem executados.

Em sistemas de tempo real, uma tarefa utiliza o processador o tempo que for necessário ou até que uma outra tarefa mais prioritária apareça.

1.4.3. Sistemas Multiprocessadores

Os sistemas multiprocessadores são caracterizados por possuir dois ou mais processadores interligados e trabalhando em conjunto. Esta característica traz com principal vantagem a possibilidade de vários programas serem executados ao mesmo tempo, de fato.

Os conceitos aplicados ao projeto de sistemas com múltiplos processadores incorporam os mesmos princípios básicos e

benefícios apresentados na multiprogramação, além de outras características e vantagens específicas como escalabilidade, disponibilidade e balanceamento de carga.

Escalabilidade é a capacidade de ampliar o poder computacional do sistema apenas adicionando novos processadores. Disponibilidade é a capacidade de manter o sistema em operação mesmo em casos de falhas. Balanceamento de carga é a possibilidade de distribuir o processamento entre os diversos processadores da configuração a partir da carga de trabalho de cada processador, melhorando, assim, o desempenho do sistema como um todo.

1.5. Estrutura do Sistema Operacional

O Sistema Operacional proporciona o ambiente pelo qual os programas são executados e é composto por um conjunto de rotinas, conhecido como o núcleo, que são disponibilizadas para as aplicações dos usuários e para o próprio sistema. A interface entre o Sistema Operacional e os programas dos usuários é definida por um conjunto de instruções que o SO proporciona. Esse conjunto de instruções é tradicionalmente chamado de chamadas de sistema (*system calls*).

O Sistema Operacional não funciona como aplicações comuns, formados de início, meio e fim. Os procedimentos do sistema são executados concorrentemente sem seguir uma ordem, com base em eventos assíncronos.

Projetar um Sistema Operacional é uma tarefa árdua e altamente importante, onde, em fase de projeto, devem-se definir todos os objetivos do sistema como um todo. As principais funções de um Sistema Operacional, presentes praticamente em todos os SO, podem ser visualizadas a seguir:

- Gerenciamento de Processos e Processador
- Gerenciamento da Memória

- Gerenciamento dos dispositivos de entrada e saída
- Gerenciamento de Arquivos

Cada parte citada pode ser uma porção bem delineada do sistema, com entradas, saídas e funções bem definidas, o que facilita o seu estudo e detalhamento. Ao longo desta apostila, nas unidades seguintes, estudaremos cada porção citada.

1.5.1. Chamadas de Sistemas

As chamadas de sistemas (*system calls*) constituem a interface entre um programa do usuário e o Sistema Operacional. Elas podem ser entendidas como uma porta de entrada para acesso ao núcleo do sistema, que contém suas funções. Sempre que o usuário necessitar de algum serviço, o solicita através de uma chamada de sistema definida e específica.

Cada serviço disponível por um determinado Sistema Operacional possui uma chamada de sistema associada e cada SO possui seu próprio conjunto de chamadas (nomes, parâmetros e formas de ativação específica).

As chamadas de sistemas, de fato, servem também para proteger o núcleo do Sistema Operacional, tentando impedir que um determinado programa realize alguma operação que altere a integridade do sistema.

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI

www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB

www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED

www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED

www.abed.org.br

Curso de Sistemas Operacionais – DCA/UFRN

Prof. Dr. Luiz Affonso Henderson Guedes de Oliveira

<http://www.dca.ufrn.br/~affonso/DCA0108/curso.html>

Home Page do Prof. Dr. Rômulo Silva de Oliveira

<http://www.das.ufsc.br/~romulo/>

Home Page do Autor Andrew S. Tanenbaum

<http://www.cs.vu.nl/~ast/>

Simulador de Ensino para Sistemas Operacionais

<http://www.training.com.br/sosim/>

REFERÊNCIAS BIBLIOGRÁFICAS

BACON, J. e HARRIS, T. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

DEITELL, H. M. & DEITEL, P. J. & CHOHNES, D. R. *Sistemas Operacionais*. São Paulo, Ed. Prentice Hall, 2005.

MACHADO, F. e MAIA, L. *Arquitetura de Sistemas Operacionais*. 4^a Edição, LTC, 2007

OLIVEIRA, R. S. e CARISSIMI, A. S. e TOSCANI, S. S. *Sistemas Operacionais*. 3^a ed, volume 11, Ed. Bookman, 2008.

SHAW, A. C. *Sistemas e software de tempo real*. Porto Alegre, Ed. Bookman, 2003.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Fundamentos de Sistemas Operacionais*. 6^a Edição, Ed. LTC.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Sistemas Operacionais com Java*. 7^a Edição, Rio de Janeiro, Ed. Elsevier, 2008.

TANENBAUM, A. S. & WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*. 3^a Edição, Porto Alegre, Ed. Bookman, 2008.

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



O conceito básico da maioria dos Sistemas Operacionais é o conceito de processos. Os processos computacionais constituem a base de administração de um sistema e, juntamente com eles, surgem vários problemas, que devem ser tratados dentro do Sistema Operacional.

Uma das principais funções de um Sistema Operacional é gerenciar os processos ativos e todos os problemas correlacionados. Dentre esses problemas correlacionados podemos destacar a concorrência entre os processos dos diversos dispositivos do sistema computacional, inclusive o processador (CPU).

Nesta unidade trataremos da parte do Sistema Operacional conhecido como gerenciador de processos. Abordaremos o conceito de processos e a estrutura do modelo de processos. Mais adiante, trataremos da comunicação entre os processos, abordando os possíveis problemas e alternativas para lidar com essa concorrência. Por fim, trataremos do gerenciamento do processador, que a partir de uma lista de processos prontos para serem executados, qual deles é escolhido para utilizar o processador.

SUMÁRIO

UNIDADE II – GERENCIAMENTO DE PROCESSOS

1. INTRODUÇÃO AO MODELO DE PROCESSOS

1.1. Conceito de Processos

1.2. Criação de Processos

1.3. Estados de um Processo

1.4. Tabelas de Processos

1.5. *Threads*

2. COMUNICAÇÃO INTERPROCESSO

2.1. Condições de Corrida

2.2. Seções Críticas

2.3. Exclusão Mútua com Espera Ativa

2.3.1. Desativando Interrupções

2.3.2. Variáveis de Bloqueio

2.3.3. Alternância Estrita

2.3.4. A Solução de Peterson

2.4. Problema dos Produtores e Consumidores

2.5. Semáforos

3. PROBLEMAS CLÁSSICOS DE CIP

3.1. Problema do Jantar dos Filósofos

3.2. Problema do Barbeiro Adormecido

4. ESCALONAMENTO DE PROCESSOS

4.1. Algoritmo de Escalonamento FIFO (*First in First out*)

4.2. Algoritmo de Escalonamento Menor Tarefa Primeiro

4.3. Algoritmo de Escalonamento *Round Robin*

4.4. Algoritmo de Escalonamento por Prioridades

4.5. Algoritmo de Escalonamento Múltiplas Filas

WEBLIOGRAFIA

REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE II

GERENCIAMENTO DE PROCESSOS

1. INTRODUÇÃO AO MODELO DE PROCESSOS

Os primeiros Sistemas Operacionais eram caracterizados por apenas um programa poder ser executado de cada vez. Os computadores mais modernos são constituídos de Sistemas Operacionais com capacidade de executar várias tarefas ao mesmo tempo.

De fato, o processador pode ser alternado de um programa para o outro, executando cada um por um determinado tempo (comumente em milissegundos). Em outras palavras, para um determinado intervalo de tempo, vários programas utilizam uma fatia desse tempo para realizar suas atividades, passando, assim, para o usuário a falsa impressão de que todos eles estão sendo executados ao mesmo tempo.

Essa falsa impressão passada ao usuário de que vários programas estão sendo executados ao mesmo tempo é comumente conhecido por **pseudoparalelismo**. Para que isso seja possível é necessário um monitoramento das múltiplas atividades entre os vários programas, que trata-se de uma tarefa difícil e bastante complexa.

Segundo *Tanenbaum*, os projetistas de Sistemas Operacionais desenvolveram um modelo que torna o paralelismo mais fácil de tratar, conhecido como **modelo de processos**, assunto deste capítulo.

1.1. Conceito de Processos

No modelo de processo, todo programa executável é organizado em um número de processos seqüenciais. Podemos definir processos como sendo a abstração do programa, ou seja, um

programa em execução, incluindo os valores do contador de programa atual, registradores e variáveis.

Em cada instante, o processador estará executando apenas um único processo, porém, como ele alterna rapidamente entre vários processos, para um determinado período de tempo, vários processos terão progredido em sua execução, passando assim a falsa impressão para o usuário que todos eles executaram em paralelo.

A diferença entre processo e programa é um ponto fundamental para o entendimento do modelo de processos. Alguns autores costumam utilizar uma analogia para facilitar esse entendimento: a receita de um bolo. Para se fazer um bolo, além da receita contendo os passos a seguir, o confeiteiro terá à sua disposição os ingredientes necessários para o preparo. Dessa forma, a receita do bolo é o programa, o confeiteiro é o processador (CPU) e os ingredientes são os dados de entrada. O processo consiste de toda a atividade de preparação do bolo, ou seja, a leitura da receita, a busca e mistura dos ingredientes, levar a massa ao forno e todas as atividades necessárias para que o bolo seja fabricado.

Agora, continuando com a analogia, imaginemos que o filho do confeiteiro apareça chorando por ter se cortado, por exemplo. O confeiteiro, no papel de um pai dedicado, deve guardar em que ponto da receita do bolo ele estava, procurar um kit de primeiros socorros e começar a seguir as orientações nele. Neste ponto, podemos verificar o processador (confeiteiro) alternando de um processo (preparo do bolo) para outro processo (cuidar do corte do filho), no qual cada um tem o seu programa próprio (receita do bolo e o livro de primeiros socorros). Quando o filho estiver tratado e medicado, o confeiteiro poderá continuar seu bolo do ponto onde parou.

1.2. Criação de Processos

Outro ponto que podemos considerar é sobre a criação de novos processos. O Sistema Operacional deve fornecer alguma maneira dos processos serem criados.

Um processo pode dar origem a diversos novos processos durante sua execução, através de uma chamada de sistema para criar um processo. Os processos criados são denominados processos filhos. O processo criador é denominado de processo pai. Cada um dos novos processos também pode criar outros processos, formando uma árvore de processos, como visualizado na Figura 6.

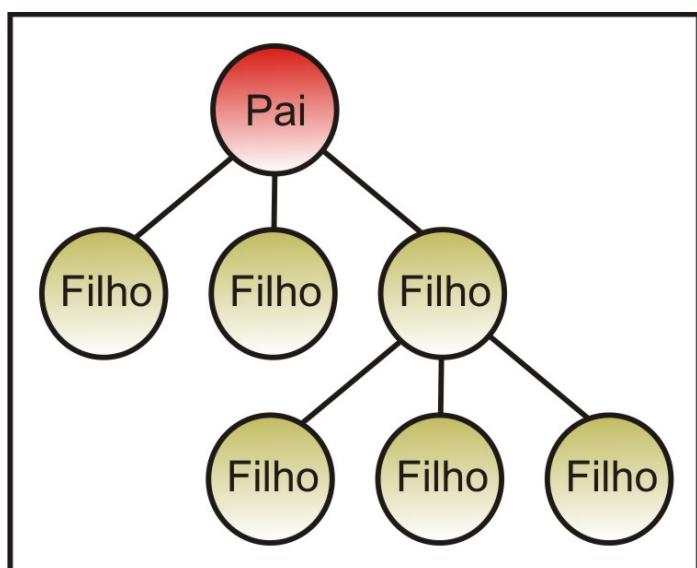


Figura 6: Árvore de Processos.

1.3. Estados de um Processo

Um processo comumente muda de estado durante sua execução. Dessa forma, um estado ativo pode estar no SO em três estados diferentes:

- **Executando:** um processo está no estado executando quando ele, de fato, está sendo processado pela CPU. Em sistemas monoprocessados (único processador), somente um processo por vez pode estar de posse da CPU em um

dado instante. Os processos se alternam na utilização do processador.

- **Pronto:** um processo está no estado de pronto quando ele possui todas as condições necessárias para a sua execução, porém, não está de posse do processador. Em geral, existem vários processos no sistema prontos para serem executados e o Sistema Operacional é responsável por, dessa lista de processos, selecionar qual utilizar o processador em um determinado instante de tempo.
- **Bloqueado:** um processo está no estado de bloqueado quando ele aguarda por algum evento externo ou por algum recurso do sistema indisponível no momento. Por exemplo, se um processo necessita de uma informação de algum dispositivo de E/S, enquanto essa informação não se torna disponível, o processo entra no estado de bloqueado.

Os três estados de um processo em um Sistema Operacional tornam possível algumas transições, como ser observado na Figura 7.

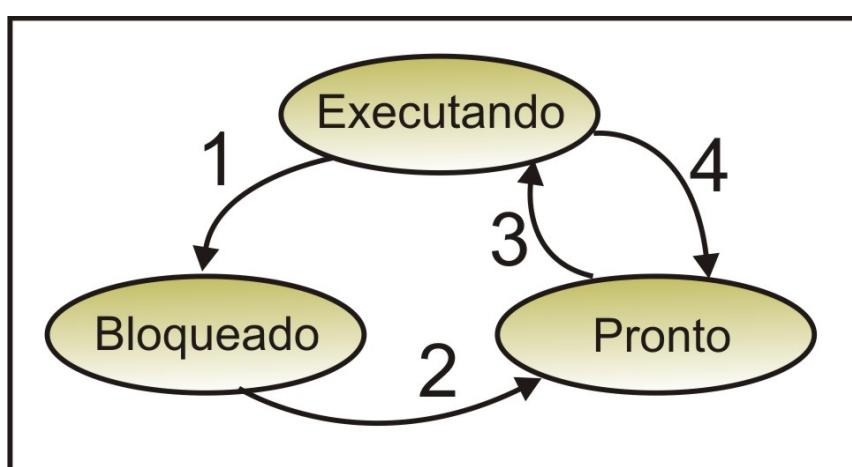


Figura 7: Transições possíveis entre os estados de um processo.

A transição 1 (Executando -> Bloqueado) ocorre quando um processo que estava utilizando o processador precisou de algum

evento externo (operação de Entrada/Saída, por exemplo), não podendo continuar executando, passando, assim, para o estado de bloqueado.

A transição 2 (Bloqueado - Pronto) ocorre quando o evento externo, no qual o processo bloqueado aguardava, acontece. Nesse caso, o processo passa para o estado de pronto e volta para a fila para poder concorrer novamente ao processador. Se não existir nenhum processo na fila de prontos, naturalmente, o processo desbloqueado utilizar a CPU.

Já as transições 3 (Pronto - Executando) e 4 (Executando - Pronto) são realizados pelo escalonador de processos. Comumente, existem vários processos prontos e esperando para serem executados. Cabe então ao Sistema Operacional (escalonador) escolher, entre os processos prontos, qual utilizará o processador e poderá executar suas atividades. O Sistema Operacional (dependendo da política do escalonador) pode, também, retirar o processador de um determinado processo e disponibilizá-lo para outro processo. Este assunto, escalonamento de processos, será tratado um pouco mais a frente, no capítulo 4 desta unidade.

1.4. Tabelas de Processos

Para ser possível a implementação do modelo de processos, o Sistema Operacional mantém uma tabela conhecida como **tabela de processos**. Essa tabela contém informações relevantes sobre os processos, como seu contador de programa, ponteiro da pilha, alocação de memória, status de arquivos abertos, dentre outros, que permite que um processo possa ser reiniciado do ponto de onde parou.

1.5. *Threads*

Threads são fluxos de execução (linha de controle) que rodam dentro de um processo. Em processos tradicionais, há uma única linha de controle e um único contador de programa. Porém,

alguns Sistemas Operacionais fornecem suporte para múltiplas linhas de controle dentro de um processo (sistemas *multithread*).

Ter múltiplas linhas de controle ou *threads* executando em paralelo em um processo equivale a ter múltiplos processos executando em paralelo em um único computador. Um exemplo tradicional do uso de múltiplas *thread* seria um navegador *web*, no qual pode ter uma *thread* para exigir imagens ou texto enquanto outro *thread* recupera dados de uma rede. É importante destacar que as *threads* existem no interior de um processo e compartilham entre elas os recursos do processo, como o espaço de endereçamento (código e dados).

2. COMUNICAÇÃO INTERPROCESSOS

Em um Sistema Operacional, frequentemente, os processos podem precisar trocar informações entre eles ou podem solicitar a utilização de um mesmo recurso simultaneamente, como arquivos, registros, dispositivos de E/S e memória. O compartilhamento de recursos entre vários processos pode causar situações indesejáveis e, dependendo do caso, gerar o comprometimento da aplicação.

O Sistema Operacional tem a função de gerenciar e sincronizar processos concorrentes, com o objetivo de manter o bom funcionamento do sistema.

2.1. Condições de Corrida

Podemos definir uma **condição de corrida** quando dois ou mais processos podem compartilhar algum espaço de memória compartilhado no qual, o resultado da informação deste espaço de armazenamento depende de quem executa e precisamente quando. Um exemplo típico de condição de corrida, apontado por vários autores, é o *spool* de impressão.

Quando um processo deseja imprimir alguma informação, ele insere o nome de arquivo em um espaço denominado diretório de *spooler*. Existe outro processo, o servidor de impressão, que verifica periodicamente se há qualquer arquivo a ser impresso e, caso haja, ele os imprime e remove a informação do diretório.

Consideremos a seguinte situação: o diretório de spooler contém um número de entradas numeradas e um processo, quando deseja imprimir alguma informação, consulta uma variável (entrada) a fim de saber em qual posição inserir o nome de arquivo no diretório. O diretório de impressão está ilustrado na Figura 8.



Figura 8: Diretório de *spooler* (impressão).

Podemos imaginar agora a seguinte situação: um processo A lê a variável entrada e armazena o valor dela (valor 0) em uma variável local. Porém, o tempo de execução do processo A termina e o Sistema Operacional o retira do processador, disponibilizando-o a um outro processo B. O processo B, por sua vez, também deseja imprimir um arquivo, acessa a área do diretório de impressão, verifica o valor da variável entrada (valor 0), armazena este valor em uma variável local e, por fim, insere o nome de seu arquivo a ser impresso na posição 0 do diretório de impressão, mudando o valor da variável entrada para 1. A Figura 9 visualiza esta situação atual.



Figura 9: Situação do Diretório de impressão após inserção do nome de arquivo do processo B.

Por fim, o processo A retoma o processador, iniciando novamente de onde parou. Ao examinar em sua variável local o valor da variável entrada (esta informação ele guardou em sua tabela, momento em que ele parou a execução), o processo observa o valor 0 e escreve seu nome de arquivo nessa posição, apagando o nome de arquivo do processo B. Em seguida, incrementa o valor da variável entrada para 1. A Figura 10 mostra a nova situação do servidor de impressão.



Figura 10: Situação do Diretório de impressão após inserção do nome de arquivo do processo A.

Internamente, o servidor de impressão continua consistente, porém o arquivo do processo B jamais será impresso. Caracterizamos este tipo de situação como uma condição de corrida.

2.2. Seções Críticas

Para se evitar uma condição de corrida é preciso definir métodos que proíba que mais de um processo acesse uma determinada área de memória compartilhada ao mesmo tempo. Esses métodos são conhecidos como **exclusão mútua**. Um processo, durante seu tempo de execução, pode realizar uma série de computações internas que não geram condições de corrida ou pode estar acessando memória compartilhada, que levam à condição de corrida.

A parte do programa no qual o processo acessa memória compartilhada é chamada **seção crítica** ou **região crítica**. Dessa forma, a solução para se evitar uma condição de corrida seria organizar os problemas de tal forma que nenhum de dois ou mais processos estivessem em suas regiões críticas ao mesmo tempo.

Para se ter uma boa solução de exclusão mútua, precisamos evitar algumas situações indesejáveis, como:

- Nenhum processo que esteja fora de sua região crítica pode bloquear a execução de outro processo;
- Nenhum processo deve esperar indefinidamente para poder entrar em sua região crítica.

Nos próximos tópicos, mostraremos algumas das diversas soluções propostas para se evitar condições de corrida.

2.3. Exclusão Mútua com Espera Ativa

Mostraremos neste tópico algumas propostas de exclusão mútua no qual, um processo quando está acessando sua região crítica, outro processo que deseja entrar também em região crítica fica aguardando.

2.3.1. Desativando Interrupções

Uma simples solução de exclusão mútua seria desativar todas as interrupções do sistema quando um processo fosse acessar sua região crítica, ativando-as imediatamente após sair dela. Com as interrupções desativas, nenhuma interrupção de relógio pode ocorrer e, naturalmente, o processador não poderá alternar entre processos, garantindo que nenhum outro processo irá executar sua região crítica.

Esta abordagem é funcional, porém, pouco atraente, pois compromete o nível de segurança do sistema. Caso as interrupções, por algum motivo, não sejam novamente ativadas, todo o sistema estará comprometido.

Outro problema deste método está em torno de sistemas com mais de um processador, pois, desativar as interrupções irá afetar apenas um processador. Com outros processadores livres, outros processos poderão executar e, ocasionalmente, acessar suas regiões críticas, tornando este método ineficaz.

2.3.2. Variáveis de Bloqueio

Outra solução de exclusão mútua seria utilizar variáveis de bloqueio (solução de *software*). A solução de variável de bloqueio consiste de uma única variável compartilhada (bloqueio). Quando um processo desejar acessar sua região crítica, ele inspeciona a variável de bloqueio. Caso a variável esteja definida com valor 0 (bloqueio livre), o processo define seu valor para 1 e acessa livremente sua região crítica. Se a variável estiver definida com valor 1, o processo deve esperar até ela se tornar 0.

Essa idéia gera o mesmo problema visto no diretório de impressão. Suponha que o processo A testa o bloqueio e verifica que ele tem o valor 0. Porém, antes de definir o seu valor para 1, outro processo é selecionado para utilizar o processador. Este novo processo testará o bloqueio, verifica que o seu valor é 0, modifica para 1 e acessa sua região crítica. O processo A, quando retomar o

processador, ele também definirá a variável de bloqueio para 1 (no momento que ele parou, ele já tinha verificado o seu valor, no momento, definido como 0) e acessa sua região crítica. Sendo assim, teremos dois processos acessando suas regiões críticas simultaneamente.

2.3.3. Alternância Estrita

Uma outra abordagem de exclusão mútua é a alternância estrita. Esta abordagem utiliza uma variável de bloqueio, que serve para indicar qual processo irá entrar em sua região crítica. Vamos analisar este método através do trecho em pseudo-linguagem apresentado a seguir:

Processo A	Processo B
<pre> <i>Enquanto(VERDADEIRO){</i> <i>Enquanto(bloqueio ≠ 0);</i> <i>//espera</i> <i>Região_crítica_A;</i> <i>Bloqueio = 1;</i> <i>Região_não_crítica_A;</i> <i>}</i> </pre>	<pre> <i>Enquanto(VERDADEIRO){</i> <i>Enquanto(bloqueio ≠ 1);</i> <i>//espera</i> <i>Região_crítica_B;</i> <i>Bloqueio = 0;</i> <i>Região_não_crítica_B;</i> <i>}</i> </pre>

Note através dos algoritmos, que quando o processo A desejar entrar em sua região crítica, primeiramente, ele testa o bloqueio. Caso o bloqueio esteja com valor 0, o processo A passa o *loop* (*enquanto*), acessando sua região crítica. Caso o processo B desejar entrar em sua região crítica e testar o bloqueio, irá verificar que o valor consta como 0 e ficará preso no laço enquanto, não podendo entrar em sua região crítica.

O processo A, ao sair da sua região crítica, irá definir o valor 1 para a variável de bloqueio, passando a executar sua região não-

crítica. Nesse ponto, o processo B terá caminho livre para acessar sua região crítica.

Agora vamos supor a seguinte situação: o processo A deseja acessar sua região crítica, testa o bloqueio e verifica que tem permissão (valor 0). O processo, assim, acessa sua região crítica, executa sua atividade, sai da região crítica, define o valor do bloqueio para 1 e começa a acessar sua região não-crítica. Nesse momento, o processo B testa o bloqueio, verifica que tem permissão, acessa sua região crítica, executa suas atividades críticas, sai da região crítica, define o bloqueio para 1 e começa a acessar sua região não-crítica. Porém, o processo B executa em sua região não-crítica rapidamente e, novamente, precisa entrar em sua região não-crítica. Porém, ele não terá acesso, pois a variável de bloqueio está com valor 0, mesmo o processo A não estando executando em sua região crítica.

Essa situação viola uma das condições para se ter uma boa solução de exclusão mútua, que define que nenhum processo que não esteja em sua região crítica pode bloquear outro processo. No caso apresentado, o processo A, mesmo não executando em sua região crítica, bloqueia o processo B, que deseja entrar em sua região crítica.

2.3.4. A Solução de Peterson

Outra maneira de se implementar exclusão mútua seria através da utilização de um algoritmo, proposto em 1981 por G. L. Peterson, conhecido como a **solução de Peterson**. Este algoritmo apresenta uma solução para dois processos que pode ser facilmente generalizada para o caso de N processos.

Este algoritmo se baseia na definição de duas primitivas, utilizadas pelos processos que desejam utilizar sua região crítica. Essas primitivas, como podem ser visualizadas no trecho de código a seguir (linguagem de programação C), são as funções `entrar_regiao()` e `sair_regiao()`, que são utilizadas pelos processos

para, respectivamente, entrarem em sua região crítica e sair de sua região crítica.

```
#define FALSO 0
#define VERDADE 1
#define N 2
int bloqueio;
int interesse[N];

void entrar_regiao(int processo){
    int outro = 1 - processo;
    interesse[processo] = VERDADE;
    bloqueio = processo;
    while(bloqueio == processo && interesse[outro] == VERDADE);
        //espera
}

void sair_regiao(int processo){
    interesse[processo] = FALSO;
}
```

Note através do código apresentado, que quando o processo A desejar entrar em sua região crítica, ele chamará a função *entrar_regiao()* passando seu identificador (int processo, com valor 0) para a função. Nesta função, a variável outro será definido com valor 1, o vetor de interesse, posição 0 (do processo), terá valor VERDADE e a variável bloqueio será definido para o valor do identificador do processo, no caso, valor 0. Ao chegar no *loop while* (enquanto), será testado se o bloqueio (valor 0) é igual ao identificador do processo e se o vetor de interesse, na posição do outro, tem valor VERDADE. No caso apresentado, o valor do vetor interesse, posição outro, não será valor VERDADE e o processo 0 não ficará preso ao *loop*, saindo da função e, naturalmente, podendo entrar em sua região crítica.

Agora vamos supor que o processo B também tem interesse de entrar em sua região crítica, chamando a função `entrar_regiao()` e passando seu identificador para a função (int processo, com valor 1). Nesse caso, ele definirá que a sua variável outro terá valor 0, que o vetor interesse, posição 1, terá valor VERDADE e a variável bloqueio será definido com o valor 1. Ao chegar no laço `while`, o processo 1 irá verificar que o vetor de interesse, na posição outro (valor 0), será VERDADE (processo 0), ficando, assim, preso no laço. Dessa forma, o processo 1 não poderá acessar sua região crítica.

O processo 0, quando sair de sua seção crítica irá chamar a função `sair_regiao()`, passando seu identificador como parâmetro para a função (int processo, com valor 0). Nesta função, o vetor de interesse, na posição do processo, receberá valor FALSO. Assim, o processo 1, que estava preso no *loop* pois o valor do vetor interesse, posição 0, tinha valor VERDADEIRO, agora poderá sair do laço e acessar sua região crítica.

2.4. Problema dos Produtores e Consumidores

O problema dos produtores e consumidores é muito comum em programas concorrentes, no qual existe um processo que produz continuamente informações que serão usadas por outro processo. Esta informação produzida e consumida pode ser um número, *string*, registro ou qualquer outro tipo. Entre o processo que produz a informação (produtor) e o processo que consome a informação (consumidor) existe um espaço de memória (*buffer*) que serve para armazenar temporariamente as informações que ainda não foram consumidas.

O problema surge quando o produtor deseja inserir um novo item, porém o *buffer* está cheio. De forma análoga, caso o consumidor deseje consumir alguma informação e o buffer estiver vazio. A solução para este problema pode ser através de duas primitivas, conhecidas como *sleep* (dormir) e *wakeup* (acordar). Para o problema dos produtores e consumidores, caso o buffer esteja

vazio, o consumidor pode ser colocado para dormir, através da primitiva *sleep* e, de forma análoga, o produtor pode ser colocado para dormir, caso o *buffer* esteja cheio. Quando uma informação for produzida, o consumidor poderá ser acordado pelo produtor através da primitiva *wakeup*. De forma análoga, o produtor pode ser acordado pelo consumidor caso uma informação seja retirada do *buffer* cheio.

Esta situação conduz à condição de corrida do servidor de impressão vista anteriormente. Supondo que é utilizada uma variável *N* para poder monitor a quantidade de informações no *buffer*. Assim, quando o valor de *N* for a quantidade máxima do *buffer*, o produtor é colocado para dormir. Da mesma forma, quando o valor de *N* for zero, o consumidor é colocado para dormir.

Agora imaginemos a seguinte situação: o *buffer* encontra-se vazio e o consumidor acabou de ler a variável *N* para ver se ele é zero. Nesse momento, o processador é tomado do consumidor e passado para o produtor. O produtor realiza sua atividade, produzindo uma informação e incrementa o valor de *N* para 1. Considerando que o valor de *N* era zero, o produtor chama a primitiva *wakeup* para acordar o consumidor (presumindo que ele estava dormindo).

Contudo, o consumidor não estava dormindo e o sinal de *wakeup* é perdido. Quando o consumidor volta a utilizar a CPU, ele testará o valor de *N* anteriormente lido, verificando que trata-se de valor zero e, por fim, passa a dormir. O produtor, cedo ou tarde, irá encher o *buffer* com informações produzidas e também irá dormir. Ambos dormirão para sempre.

2.5. Semáforos

O problema dos produtores e consumidores, além dos vários problemas de sincronização entre processos, pode ser resolvido através de uma solução conhecida por **semáforos**. Ele foi proposto por *E. W. Dijkstra* em 1965. O semáforo é um tipo abstrato de dado composto por um valor inteiro e uma fila de processos. *Dijkstra*

propôs existir duas operações básicas sobre semáforos: operação de testar e a operação de incrementar. Vários autores utilizam nomenclaturas diferentes para representar essas operações: DOWN e UP (*Tanenbaum*), *wait* e *signal* (*Silberschatz*) e P e V, originalmente designadas, que vem do holandês *proberen* (testar) e *verhogen* (incrementar).

Quando um processo executa a operação P sobre um semáforo, o seu valor inteiro é decrementado. Se o valor do semáforo for negativo, o processo é bloqueado e inserido no fim da fila desse semáforo. Quando um processo executa a operação V sobre um semáforo, o seu valor inteiro é incrementado. Caso exista algum processo bloqueado na fila, o primeiro processo é liberado.

Para que a solução através de semáforos funcione corretamente suas operações são feitas como uma única e indivisível **ação atômica**, ou seja, uma vez que uma operação sobre semáforos comece, ela não pode ser interrompida no meio e nenhuma outra operação sobre o semáforo deve ser começada.

Dessa forma, para utilizar a solução em uma aplicação que gere uma condição de corrida basta que, para cada estrutura de dados compartilhada, seja utilizado um semáforo. Todo processo antes de acessar essa estrutura deve realizar uma operação de teste do semáforo (P). Ao sair da seção crítica, o processo deve realizar uma operação V sobre o semáforo.

3. PROBLEMAS CLÁSSICOS DE CIP

A literatura de Sistemas Operacionais está repleta de problemas que servem como ambiente de aplicação de novos métodos de comunicação interprocessos. Segundo Tanenbaum, todo mundo que resolia inventar algum método para solucionar problemas de sincronização entre processos sentia-se obrigado a

demonstrar o quão maravilhoso o seu método era, aplicando sobre um desses problemas clássicos e mostrando o quão elegantemente seu método resolia. Neste tópico, examinaremos dois problemas clássicos de comunicação interprocessos conhecidos como: Problema do jantar dos filósofos e o problema do barbeiro adormecido.

3.1. Problema do Jantar dos Filósofos

O problema do Jantar dos Filósofos foi proposto e resolvido por Dijkstra em 1965. O problema foi modelado da seguinte forma:

- Cinco filósofos sentados ao redor de uma mesa circular, que contém cinco pratos de espaguete e cinco garfos.
- Entre cada prato de espaguete encontra-se um garfo.
- Para comer, um filósofo precisa de dois garfos.

O problema do jantar dos filósofos pode ser ilustrado através da Figura 11.

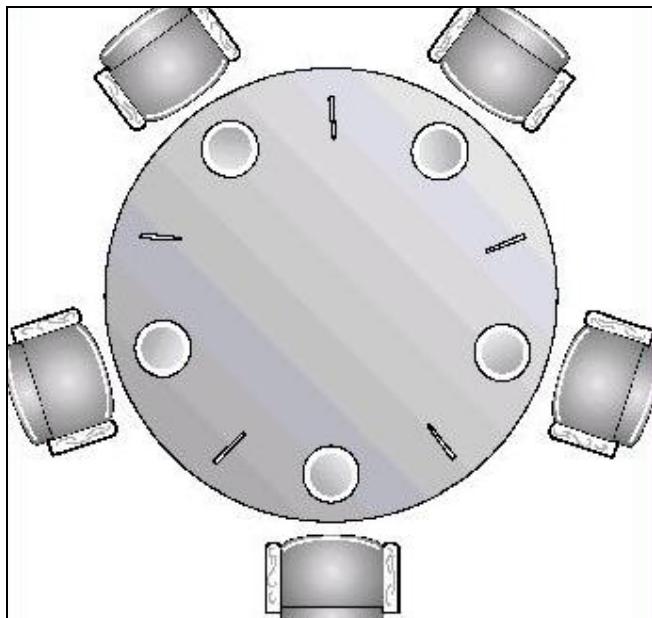


Figura 11: O problema clássico de comunicação interprocessos do Jantar dos Filósofos.



Segundo o problema, cada filósofo alterna períodos de comer e pensar. Assim, quando um filósofo fica com fome, ele tentar pegar os garfos da sua esquerda e direita, um de cada vez, para poder comer o espaguete. Se conseguir pegar os dois garfos, ele come por um tempo, liberando os garfos ao final e voltando à sua atividade de pensar.

O problema consiste em modelar uma situação que não gere o bloqueio dos filósofos. Vamos imaginar, por exemplo, a seguinte situação: suponha que todos os filósofos resolvem comer ao mesmo tempo e pegam seus garfos da direita. Nessa situação, nenhum terá condições de pegar o garfo da esquerda (indisponível) e entraram em uma situação de impasse.

Uma outra situação que poderíamos imaginar para resolver o problema seria fazer com que o filósofo depois que pegasse o garfo da direita, verificar se o garfo da esquerda está disponível. Se estiver disponível, o filósofo poderia pegar e comer. Caso não estivesse disponível, o filósofo colocaria o garfo da direita e esperaria algum tempo para, então, tentar comer novamente. Esta proposta também geraria uma situação de impasse entre os filósofos. Imagine se todos resolvessem comer ao mesmo tempo, pegassem seus garfos da direita e, após verificar que os garfos da esquerda não estavam disponíveis, baixar o garfo da direita, esperar um tempo qualquer e todos, mais uma vez, resolvessem comer novamente.

Poderíamos pensar em uma solução onde o tempo que o filósofo esperaria para comer novamente fosse aleatório. Em geral, poderia funcionar, mas teríamos, ainda, uma probabilidade de que o tempo aleatório de cada um fosse o mesmo, gerando, assim, a situação de impasse. Uma provável solução para este problema pode ser pensada utilizando semáforos.

3.2. Problema do Barbeiro Adormecido

Outro problema clássico de CIP é conhecido como o **problema do barbeiro adormecido**. Trata-se de uma barbearia que

contém um barbeiro, uma cadeira de barbeiro e n cadeiras de espera. O problema pode ser visualizado através da Figura 12.

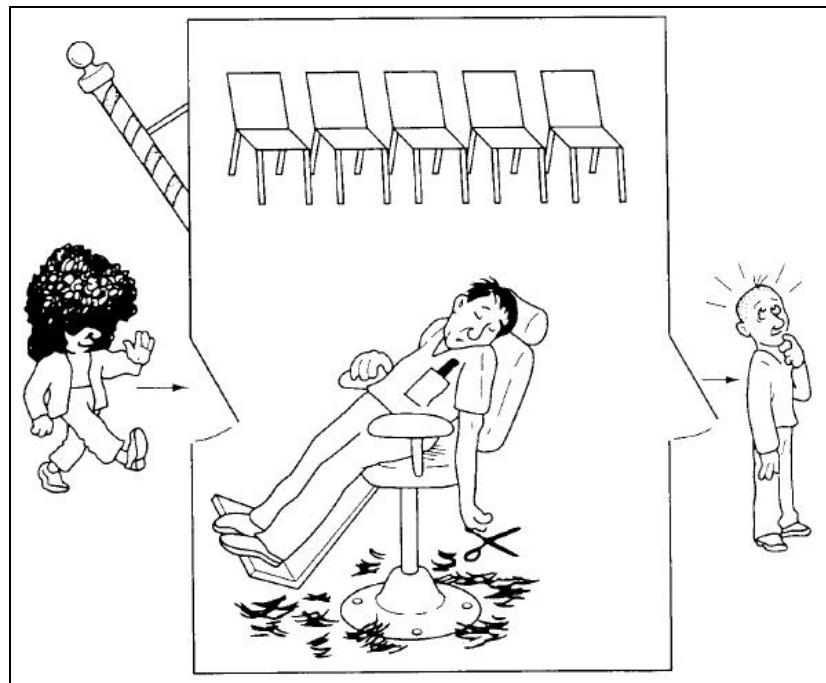


Figura 12: O problema do Barbeiro Adormecido.

Para este problema, se não houver nenhum cliente na barbearia, o barbeiro senta-se e começa a dormir (o barbeiro, provavelmente, não dormiu direito durante a noite). Quando um cliente chega à barbearia, ele acorda o barbeiro para poder cortar seu cabelo. Se outros clientes chegarem enquanto o barbeiro estiver cortando o cabelo de alguém, o novo cliente observa se existe cadeira de espera livre. Caso exista, ele se senta e aguarda sua vez. Caso contrário, ele é obrigado a ir embora.

Note que neste problema temos várias áreas compartilhadas e qualquer método que tente resolver este problema deverá considerar esses pontos.

4. ESCALONAMENTO DE PROCESSOS

Um ponto crucial no desenvolvimento de um Sistema Operacional é como alocar de forma eficiente o processador (CPU) para os vários processos prontos para serem executados. O **escalonamento de processos** é a forma com que os processadores disponíveis (um ou vários) no Sistema Computacional são distribuídos ou alocados para os vários processos prontos.

Segundo *Silberschatz*, o objetivo da multiprogramação é contar sempre com algum processo em execução para maximizar a utilização da CPU. Em um sistema com um único processador, somente um processo pode ser executado de cada vez; quaisquer outros processos devem esperar até que a CPU esteja livre e possa ser redistribuída.

A parte do Sistema Operacional responsável por selecionar qual será o processo que executará no processador é chamado de **escalonador** ou **agendador**. Dessa forma, a ordem com que os processos serão executados pelo processador é definida por um determinado **algoritmo ou política** de escalonamento de processos.

O projeto de um algoritmo de escalonamento deve levar em conta uma série de necessidades, no qual alguns são apontados a seguir:

- Utilização da CPU: o intuito é manter a CPU ocupada o tempo máximo possível.
- Maximizar a produtividade (*throughput*): se a CPU estiver ocupada executando processos, então o trabalho estará sendo realizado. Deve-se procurar maximizar o número de processos processados por unidade de tempo.
- Justiça: o algoritmo de escalonamento deve ser justo com todos os processos, onde cada um deve ter uma chance de usar o processador.

- Minimizar o tempo de resposta: intervalo de tempo entre a submissão de uma solicitação e o momento em que a primeira resposta é produzida.

Outras necessidades podem ser apontadas, como tempo de espera, balanceamento do uso dos recursos, previsibilidade, dentre outros. Se observarmos, alguns desses objetivos de um algoritmo de escalonamento de processos são conflitantes. Além disso, os processos são únicos e imprevisíveis. Dessa forma, o grande desafio de se desenvolver algum algoritmo de escalonamento é conseguir balancear todos esses objetivos conflitantes. Porém, segundo pesquisadores da área, qualquer algoritmo de escalonamento que tenta favorecer alguma classe de trabalho (sistemas interativos, por exemplo) acaba prejudicando outra classe de trabalho, ou seja, para dar mais a um usuário, você tem que dar menos a outro, como dito por *Tanenbaum*.

Os algoritmos de escalonamentos podem ser classificados em *preemptíveis* e *não-preemptíveis*. O algoritmo de escalonamento é *não-preemptível* quando o processador alocado para um determinado processo não pode ser retirado deste até que o processo seja finalizado. Já o algoritmo de escalonamento é dito *preemptível* quando o processador alocado para um determinado processo pode ser retirado deste em favor de um outro processo. Este procedimento de tirar o processador de um processo e atribuí-lo à outro é chamado por vários autores da área por **troca de contexto**.

Nos tópicos a seguir, veremos com detalhes alguns algoritmos de escalonamento de processos.

4.1. Algoritmo de Escalonamento FIFO (*First in First out*)

Trata-se do algoritmo de escalonamento de implementação mais simples. Com este algoritmo de escalonamento, o primeiro processo que solicita a CPU é o primeiro a ser alocado. Dessa forma, os processos que estão prontos para serem executados pela

CPU são organizados numa fila, que funciona baseado na política FIFO (First in First out – Primeiro a entrar é o primeiro a sair).

Vejamos um exemplo. Considere o seguinte conjunto de processos:

Processo	Duração de Execução
A	12
B	8
C	15
D	5

Supondo que a ordem de chegada dos processos seja: A – B – C – D. Dessa forma, baseado na política FIFO, a ordem de execução dos processos é mostrado através da Figura 13 (diagrama de tempo).

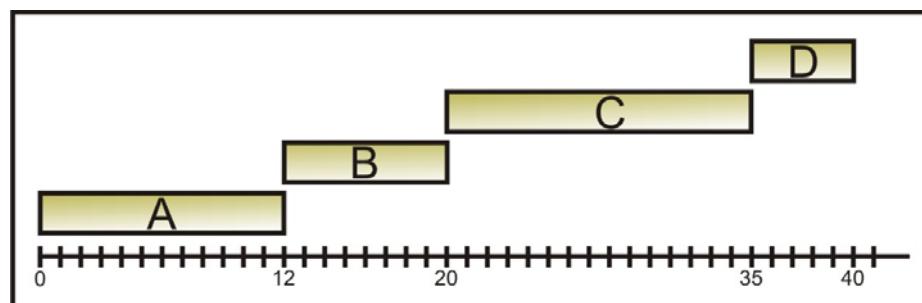


Figura 13: Diagrama de tempo usando a política FIFO.

Para este conjunto de tarefas o tempo de espera do processo A é de 0 (zero) unidades de tempo; para o processo B de 12 unidades de tempo; para o processo C de 20 unidades de tempo; e para o processo D de 35 unidades de tempo. O tempo médio de espera na fila de prontos é de $(0+12+20+35)/4$, que equivale a 16,75 unidades de tempo.

Nesta política de escalonamento o tempo médio de espera é, com frequência, um tanto longo. Outro ponto é que processos importantes podem ser obrigados a esperar devido à execução de outros processos menos importantes dado que o escalonamento

FIFO não considera qualquer mecanismo de distinção entre processos.

4.2. Algoritmo de Escalonamento Menor Tarefa Primeiro

Outra política de escalonamento de tarefas é conhecida como algoritmo de escalonamento de Menor Tarefa Primeiro (SJF – *shortest job first*), no qual o processo que tem o menor ciclo de processamento (tempo de execução) será selecionado para usar o processador.

Considerando o mesmo conjunto de tarefas apresentados, teríamos o diagrama de tempo apresentado na Figura 14.

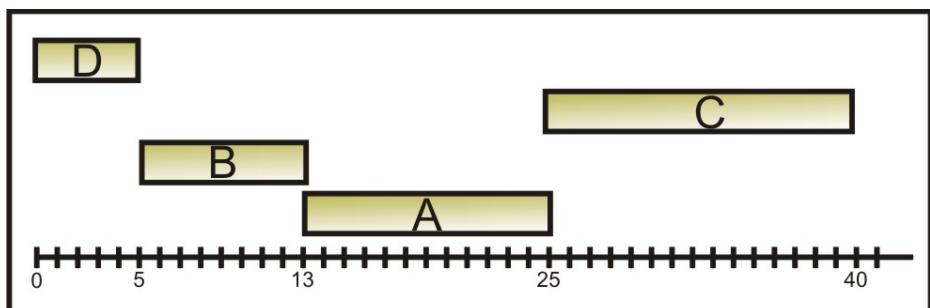


Figura 14: Diagrama de tempo usando a política Menor Tarefa Primeiro.

Nesta política de escalonamento, o tempo de espera do processo A é de 13 unidades de tempo; para o processo B de 5 unidades de tempo; para o processo C de 25 unidades de tempo; e para o processo D de 0 unidades de tempo. O tempo médio de espera na fila de prontos é de $(13+5+25+0)/4$, que equivale a 10,75 unidades de tempo. Em média, nessa política de escalonamento, os processos tiveram que esperar menos para serem executados pelo processador.

Segundo *Silberschatz*, a dificuldade real com o algoritmo de Menor Tarefa Primeiro é saber o tempo de duração da próxima solicitação de CPU. Assim, trata-se de um algoritmo ótimo porém, não pode ser implementado, pois não há modo de saber o tempo de duração do próximo pico de CPU. Uma abordagem possível é tentar aproximar-se do algoritmo de Menor Tarefa Primeiro.

4.3. Algoritmo de Escalonamento *Round Robin*

O algoritmo *Round Robin*, conhecido também como algoritmo de escalonamento circular, também organiza a lista de processos prontos como uma fila simples, semelhante ao algoritmo FIFO. No entanto, cada processo recebe uma fatia de tempo do processador, comumente chamado de *quantum*.

Assim, um processo executa durante um *quantum* específico. Se o *quantum* for suficiente para este processo finalizar, outro processo da fila é selecionado para executar. Se durante sua execução o processo se bloquear (antes do término do *quantum*), outro processo da fila de prontos também é selecionado. Se terminar a fatia de tempo do processo em execução, ele é retirado do processador, que é disponível para outro processo. Note então, que trata-se de um algoritmo de escalonamento *preemptível*.

Vamos considerar o mesmo conjunto de tarefas apresentado nesta apostila na seção correspondente ao algoritmo de escalonamento FIFO. Para escalaronar este conjunto de tarefas utilizando o algoritmo de escalonamento *Round Robin*, com *quantum* de 4 unidades de tempo, teríamos o diagrama de tempo representado através da Figura 15.

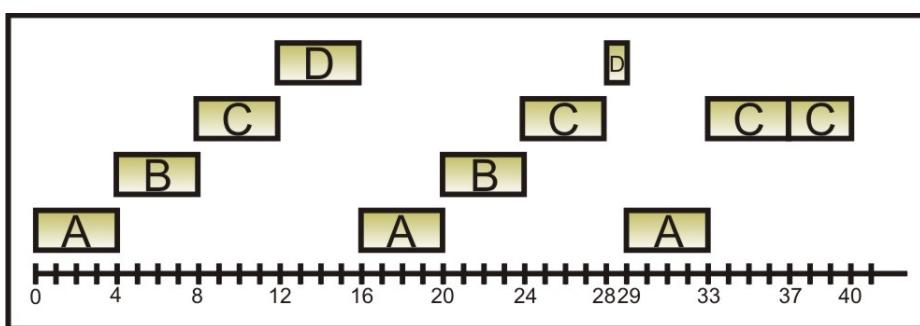


Figura 15: Diagrama de tempo usando a política *Round Robin* (quantum 4 unidades de tempo).

Uma grande questão relacionada à política de escalonamento *Round Robin* é a definição do *quantum*. A troca de contexto (alternar entre um processo e outro) requer certa quantidade de tempo para

ser realizada. Sendo assim, se o *quantum* definido for muito pequeno, ocasiona uma grande quantidade de trocas de processos e a eficiência da CPU é reduzida; de forma análoga, a definição do *quantum* muito grande pode tornar a política *Round Robin* numa FIFO comum.

4.4. Algoritmo de Escalonamento por Prioridades

Outra política de escalonamento bem interessante é a política por prioridades. Nesta política, os processos são organizados na fila de prontos baseado em prioridades. Quem tiver maior prioridade vai para o início da fila. Quem tiver menor prioridade vai se encaixando no final da fila. Esta prioridade pode ser uma atribuição externa ao sistema.

Vejamos um exemplo. Considere o seguinte conjunto de processos:

Processo	Prioridade	Duração de Execução
A	2	10
B	4	8
C	3	6
D	1	4

Dessa forma, baseado na política de escalonamento por prioridades (quanto menor o número, maior a prioridade), a ordem de execução dos processos é mostrado através da Figura 16 (diagrama de tempo).

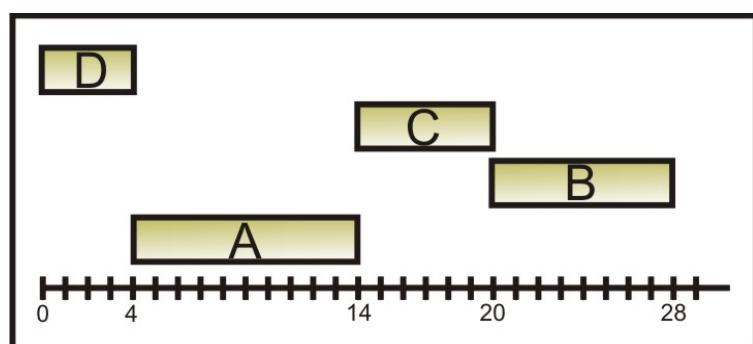


Figura 16: Diagrama de tempo usando a política por prioridades.

Alguns aspectos devem ser considerados na política de escalonamento por prioridades. Primeiro, se no sistema existir uma quantidade grande e interativa de processos de alta prioridade, podemos chegar a uma situação onde processos de baixa prioridade nunca executarão.

Uma possível solução para este problema é a utilização de prioridades dinâmicas. Dessa forma, os processos de baixa prioridade podem ter suas prioridades lentamente aumentadas, tendo, assim, chances de utilizar o processador.

Segundo Oliveira, tipicamente, soluções baseadas em prioridades utilizam preempção, pois o mecanismo de preempção permite implementar o conceito de prioridades de uma maneira mais completa no sistema, posto que não faz sentido fazer todo um esforço para executar antes processos com prioridades alta e, ao mesmo tempo, permitir que um processo de baixa prioridade ocupe o processador indefinidamente (caso não haja preempção).

4.5. Algoritmo de Escalonamento Múltiplas Filas

Comumente, em um Sistema Operacional, existem vários processos de mesmo tipo (mesma categoria, baseado na prioridade e consumo de recursos). Dessa forma, ao invés de termos apenas uma única fila de prontos, poderíamos construir várias filas de prontos e agrupar os processos de mesma categoria nessas filas. Para cada fila poderíamos definir prioridades diferentes e políticas de escalonamentos específicas. Este tipo de algoritmo de escalonamento é conhecido como algoritmo de **Múltiplas Filas**.

Um exemplo deste algoritmo seria considerar duas filas de prontos, uma com maior prioridade e outra de menor prioridade, as duas funcionando segundo a política *Round Robin*. Dessa forma, se a fila mais prioritária tiver processos, estes terão prioridade sobre o processador. Caso a fila mais prioritária estiver vazia, os processos prontos da fila menos prioritária irão concorrer ao processador.

Note que a partir dos vários algoritmos apresentados, é possível desenvolver vários outras combinações e estratégias de escalonamento. Segundo Oliveira, tipicamente, a maioria dos sistemas trabalham com fatia de tempo, sendo muito comum utilizar prioridades para favorecer determinados processos que realizam tarefas para o próprio Sistema Operacional.

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI

www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB

www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED

www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED

www.abed.org.br

Curso de Sistemas Operacionais – DCA/UFRN

Prof. Dr. Luiz Affonso Henderson Guedes de Oliveira

<http://www.dca.ufrn.br/~affonso/DCA0108/curso.html>

Home Page do Prof. Dr. Rômulo Silva de Oliveira

<http://www.das.ufsc.br/~romulo/>

Home Page do Autor Andrew S. Tanenbaum

<http://www.cs.vu.nl/~ast/>

Simulador de Ensino para Sistemas Operacionais

<http://www.training.com.br/sosim/>

REFERÊNCIAS BIBLIOGRÁFICAS

BACON, J. e HARRIS, T. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

DEITELL, H. M. & DEITEL, P. J. & CHOHNES, D. R. *Sistemas Operacionais*. São Paulo, Ed. Prentice Hall, 2005.

MACHADO, F. e MAIA, L. *Arquitetura de Sistemas Operacionais*. 4^a Edição, LTC, 2007

OLIVEIRA, R. S. e CARISSIMI, A. S. e TOSCANI, S. S. *Sistemas Operacionais*. 3^a ed, volume 11, Ed. Bookman, 2008.

SHAW, A. C. *Sistemas e software de tempo real*. Porto Alegre, Ed. Bookman, 2003.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Fundamentos de Sistemas Operacionais*. 6^a Edição, Ed. LTC.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Sistemas Operacionais com Java*. 7^a Edição, Rio de Janeiro, Ed. Elsevier, 2008.

TANENBAUM, A. S. & WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*. 3^a Edição, Porto Alegre, Ed. Bookman, 2008.

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



Resumo

Uma das principais funções de um Sistema Operacional é controlar e gerenciar todos os dispositivos de entrada e saída disponíveis no Sistema de Computação. Os dispositivos de entrada e saída são responsáveis por fazer a comunicação do sistema em si com o usuário ou meio externo. O Sistema Operacional deve ser responsável por fazer a interface entre os dispositivos em si e o resto do Sistema Computacional, enviando comandos para os dispositivos, capturando interrupções e tratando possíveis erros.

Além disso, o Sistema Operacional deve apresentar uma interface amigável para o usuário, escondendo toda complexidade de *hardware*, típico dos dispositivos de entrada e saída.

Nessa unidade trataremos da parte do Sistema Operacional responsável por fazer toda essa tarefa descrita, que é chamada de **gerenciamento de dispositivos de entrada e saída** ou, simplesmente, gerenciamento de entrada e saída. Trataremos também na unidade os impasses causados por tentativas de acesso concorrente aos vários dispositivos disponíveis no Sistema Computacional.

SUMÁRIO

UNIDADE III – GERENCIAMENTO DE ENTRADA E SAÍDA

1. INTRODUÇÃO
 2. PRINCÍPIOS DE *HARDWARE* DE E/S
 - 2.1. Dispositivos de Entrada e Saída
 - 2.2. Controladoras de Dispositivos
 - 2.2.1. Controladoras que suportam Acesso Direto à Memória
 3. PRINCÍPIOS DE *SOFTWARE* DE E/S
 - 3.1. Manipuladores de Interrupções
 - 3.2. *Drivers* de Dispositivos
 - 3.3. *Software* de E/S Independente de Dispositivo
 - 3.4. *Software* de E/S de Nível de Usuário
 4. DEADLOCKS
 - 4.1. Condições para um Impasse
 - 4.2. Modelagem de Impasses através de Grafos
 - 4.3. Métodos de Lidar com *Deadlocks*
 - 4.3.1. Algoritmo do Avestruz
 - 4.3.2. Detecção e Recuperação
 - 4.3.3. Prevenção de Impasses
 - 4.3.4. Impedimento de Impasses
- WEBLIOGRAFIA
- REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE III

GERENCIAMENTO DE ENTRADA E SAÍDA

1. INTRODUÇÃO

Uma das principais funções de um Sistema Operacional, sem sombra de dúvida, é gerenciar e controlar todos os dispositivos de entrada e saída disponíveis. Esta tarefa não é nada simples, posto que o Sistema Operacional deve, para realizar esta atividade, enviar comandos para os dispositivos, capturar e enviar interrupções, além de tratar possíveis erros.

O computador, de fato, se tornaria um equipamento inútil se não existisse alguma forma do meio externo se comunicar com ele. Os dispositivos de entrada e saída (E/S) são responsáveis por fazer essa comunicação entre o sistema, de modo geral, e o meio externo.

Nesta unidade começaremos abordando os princípios de *hardware* de E/S. Logo em seguida, trataremos dos princípios de *software* de E/S, ou seja, como o *software* que controla e faz a comunicação com um dispositivo é organizado. Esta organização é mostrada na Figura 17.

Mais adiante, trataremos da concorrência entre os processos para alocar os dispositivos em geral e como esse acesso pode gerar uma situação de impasse.

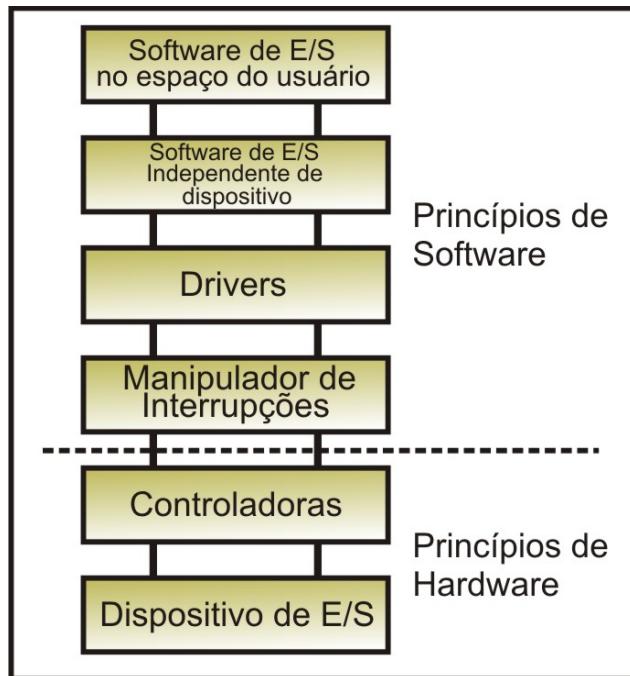


Figura 17: Princípios de *hardware* e de *software* de E/S.

2. PRINCÍPIOS DE *HARDWARE* DE E/S

Segundo *Tanenbaum*, o *hardware* de entrada e saída pode ser visto sob dois pontos de vista:

- Visão do Engenheiro Elétrico: vê o *hardware* sob o ponto de vista de chips, fios, fontes de alimentação, de motores e de todos os componentes físicos que o constituem.
- Visão do Programador: já o programador vê a interface apresentada para o software (comandos que o *hardware* aceita, as funções que ele executa e os erros que podem ser retornados).

Nesta apostila, como no estudo de Sistemas Operacionais baseado no livro clássico de Sistemas Operacionais Modernos de *Andrew Tanenbaum*, abordaremos a programação de dispositivos de

E/S e não o seu funcionamento interno (assunto de maior interesse dos engenheiros elétricos).

2.1. Dispositivos de Entrada e Saída

Como já dito anteriormente, o dispositivo de entrada e saída é o mecanismo utilizado para fazer a interface entre o mundo exterior e o sistema e sem ele o computador não teria muita funcionalidade. É possível encontrar uma grande diversidade de dispositivos de entrada e saída, como: teclados, mouses, monitor de vídeo, impressora, scanners, dentre outros.

Os dispositivos de entrada e saída, dependendo do sentido do fluxo de informações entre o sistema e o dispositivo, podem ser divididos, grosso modo, em dispositivos de entrada, dispositivos de saída ou dispositivos de entrada e saída. Os dispositivos de entrada são caracterizados por conter um fluxo de informações do dispositivo para o sistema, ou seja, são responsáveis por inserir no sistema informação do mundo externo. Já os dispositivos de saída são caracterizados pelo fluxo de informações do sistema para o mundo externo, ou seja, responsáveis por disponibilizar respostas ao mundo externo. Já os dispositivos e entrada e saída contemplam os dois fluxos.

Segundo *Tanenbaum*, os dispositivos também podem ser divididos, grosso modo, em duas categorias: **dispositivos de bloco** e **dispositivos de caractere**. Os dispositivos de bloco são caracterizados por armazenar informações em blocos de tamanhos fixos, cada um com seu endereço próprio. A propriedade essencial desse tipo de dispositivos é que é possível ler ou gravar blocos independentemente um do outro. O disco é um bom exemplo deste tipo de dispositivo.

Já os dispositivos de caractere são caracterizados por aceitar ou entregar um fluxo de caracteres, sem considerar qualquer estrutura de bloco, sem endereçamento ou qualquer operação de busca.

Embora esta classificação seja, de forma geral, muito utilizada, segundo o autor *Tanenbaum*, existem dispositivos que não são classificáveis em nenhum desses tipos. Como exemplo, tomemos por base os relógios. Os relógios são dispositivos que geram interrupções em intervalos definidos de tempo. Este dispositivo não se encaixa nos dispositivos de bloco, pois não possuem estrutura de blocos e, também, não se encaixam como dispositivos de caractere, pois não gera nenhum fluxo de caracteres.

2.2. Controladoras de Dispositivos

Para que os dispositivos se comuniquem com o sistema, eles são ligados ao computador através de um componente de *hardware* chamado de **interface**. Assim, os dispositivos não estão ligados diretamente aos barramentos do computador. Devido a diversidade de tipos de dispositivos, que abstrai diferentes formas de operações e complexidade, as interfaces empregam no seu projeto um outro componente de *hardware* conhecido como **controladora de dispositivo**.

A controladora de dispositivo (chamada também de **adaptador de dispositivo**) trata-se de um componente eletrônico, comumente na forma de uma placa de circuito impresso, que pode ser inserido na placa mãe do computador. O dispositivo em si trata-se de um componente mecânico. Uma controladora pode manipular mais de um dispositivo e, quando padronizadas, podem ser fabricadas por diversas empresas. Como exemplo de controladoras, temos as controladoras de disco IDE ou SCSI.

Cada controladora deve saber especificamente como o dispositivo a ela relacionado funciona, a fim de que possa enviar comandos a serem realizados. Basicamente, uma controladora tem a função de converter um fluxo serial de bits em um bloco de bytes e executar uma correção de erros. Este bloco de bytes, montado em um *buffer* interno da controladora, após verificado erros, pode ser copiado para a memória principal. A maioria dos Sistemas



Exemplo de um disco com interface IDE e SCSI, respectivamente

Operacionais quase sempre lidam com a controladora, não com o dispositivo.

As controladoras são formadas de um conjunto de registradores que são enxergados pelo processador. Esses registradores recebem ordens do processador, fornecem o estado de uma operação ou permitem a leitura ou escrita de dados do dispositivo. O Sistema Operacional quando deseja dar algum comando à controladora acessam esses registradores (cada registrador possui um endereço). Os endereços dos registradores podem fazer parte do espaço normal de endereçamento de memória. Esse esquema é conhecido como **mapeamento de entrada e saída em memória**.

2.2.1. Controladoras que suportam Acesso Direto à Memória

As controladoras comuns (sem acesso direto à memória) recebem um fluxo de caracteres, converte para um bloco, armazenado em seu *buffer*, depois verifica possíveis erros. Por seguite, a controladora gera uma interrupção no Sistema Operacional, que por sua vez, começa a ler o bloco do buffer da controladora, cada byte por vez, e armazena-os na memória principal. Esta operação, naturalmente, exige desperdício de CPU.

O **Acesso Direto à Memória (DMA – Direct Memory Access)** é um mecanismo criado para liberar a CPU do trabalho de cópia dos dados da controladora para a memória. O controlador DMA é conectado diretamente ao barramento de dados e de endereços do computador, para ter a capacidade de acessar diretamente endereços de memória.

Com a controladora DMA, após o bloco ter sido lido do dispositivo completamente e verificado possíveis erros, ela copia o primeiro byte para o endereço de memória principal especificado pelo endereço de memória DMA. Após terminar a transferências dos dados para a memória principal, a controladora DMA gera uma interrupção para o Sistema Operacional, que ao ser iniciado já encontra o dado em memória principal.

Segundo Oliveira, a técnica de DMA é mais eficiente quando a operação de entrada e saída envolve a leitura ou escrita de muitos dados, como exemplo, uma leitura de disco. Nem todos os computadores utilizam a técnica de DMA. A justificativa disso é o fato de que a CPU, comumente, é bem mais rápida que a controladora DMA e pode fazer o trabalho de cópia de dados para a memória muito mais rápida.

3. PRINCÍPIOS DE SOFTWARE DE E/S

O *software* de entrada e saída, comumente, é organizado em uma estrutura de camadas, no qual as camadas mais baixas têm como principal função esconder das camadas mais altas as peculiaridades do hardware. Já as camadas mais altas têm como principal função apresentar uma interface amigável e simples para o usuário final.

Algumas metas de *software* de entrada e saída podem ser apontadas:

- Independência de dispositivo: deve ser possível escrever programas que podem, por exemplo, lê arquivos de qualquer dispositivo.
- Atribuição uniforme de nomes: o nome de um arquivo ou de um dispositivo não pode depender do dispositivo.
- Tratamento de erros: os erros devem ser tratados o mais perto possível do *hardware*.
- Transferências síncronas ou assíncronas: a maior parte dos dispositivos de E/S são assíncronos e os programas dos usuários são mais fáceis de implementar através de bloqueios. Assim o Sistema Operacional deve fazer com que as operações pareçam, de fato, com bloqueios para os programas do usuário.

- Dispositivos compartilháveis ou dedicados: O Sistema Operacional deve ser capaz de tratar dispositivos tanto dedicados como compartilhados de uma maneira que não gere problemas.

Trataremos neste tópico quatro camadas de *software* de entrada e saída: manipuladores de interrupções, os *drivers* de dispositivos, *software* de E/S independente de dispositivo e *software* de E/S de nível de usuário.

3.1. Manipuladores de Interrupções

As interrupções devem ser escondidas do sistema. O ideal é bloquear um processo que inicia uma operação de E/S e mantê-lo bloqueado até que a operação de E/S complete e a interrupção tenha ocorrido. O importante a se saber é que quando uma interrupção é gerada, o processo é desbloqueado, passando para um estado capaz de ser executado.

3.2. Drivers de Dispositivos

O *driver* de dispositivo (*device driver*) é composto de um conjunto de módulos de software cada um implementado para fornecer mecanismos de acesso a um dispositivo de entrada e saída específico. Assim, cada *driver* de dispositivo trata de um tipo de dispositivo ou de uma classe de dispositivos correlacionados.

De modo geral, um driver é responsável por aceitar uma solicitação abstrata (por exemplo, ler alguma informação) e cuidar para que esta solicitação seja atendida. Assim o *driver* de um dispositivo tem o conhecimento de como funciona a controladora que o dispositivo usa.

3.3. Software de E/S Independente de Dispositivo

A principal função do *software* de E/S independente de dispositivo é executar funções que são comuns para vários dispositivos.

dispositivos e oferecer uma interface uniforme para o *software* de nível de usuário.

Segundo Oliveira, podemos enumerar alguns serviços sob responsabilidade dessa camada:

- **Nomeação de Dispositivo:** cada dispositivo deve receber um nome lógico e ser identificado a partir dele.
- **Buferização:** buffer é uma zona de memória temporária utilizada para armazenar dados enquanto eles estão sendo transferidos entre as diferentes camadas do *software* de E/S.
- **Cache de Dados:** armazenar na memória um conjunto de dados que estão sendo frequentemente utilizados.
- **Alocação e Liberação:** devido a alguns dispositivos admitirem, no máximo, um usuário por vez, o *software* de E/S deve gerenciar a alocação, liberação e uso destes dispositivos.
- Tratamento de Erros: o software de E/S deve fornecer mecanismos de manipulação de erros, informando à camada superior o sucesso ou fracasso de uma operação.

3.4. Software de E/S de Nível de Usuário

A visão dos dispositivos de E/S para o usuário consiste de bibliotecas vinculadas em programas de usuários. Essas bibliotecas são fornecidas pelas linguagens de programação e podem ser utilizadas pelos usuários e ligadas com seus programas para compor o arquivo executável.

As funções de entrada e saída são dependentes e específicas de cada linguagem de programação. Por exemplo, na linguagem C as funções *printf* e *scanf* são utilizadas para impressão formatada e leitura, respectivamente. As bibliotecas de entrada e saída não fazem parte do núcleo do Sistema Operacional. Elas são associadas às várias linguagens de programação.

4. DEADLOCKS

O Sistema Computacional está repleto de recursos, que podem ser utilizados pelos processos. Comumente, a quantidade de recursos disponíveis é muito menor que a quantidade de processos solicitando esse recurso. Dependendo do recurso, este fato pode gerar uma situação de bloqueio eterno do processo, o que é muito ruim para os sistemas.

Podemos, assim, definir que um conjunto de processos está em **deadlock** ou **impasse** quando cada um desses processos está bloqueado esperando um evento que só pode ser gerado por um outro processo desse conjunto. Como todos os processos estão bloqueados, nenhum deles poderá gerar um evento e, naturalmente, todos continuarão nessa situação.

Um bom exemplo apontado por Tanenbaum trata-se de dois processos querendo ler dados de um CD e imprimindo em uma impressora. Suponha que o processo A solicite a impressora, que é disponibilizada. De forma análoga, o processo B solicita o CD-ROM, que lhe é disponibilizado. Agora o processo A solicita o CD-ROM, enquanto que o processo B solicita a impressora. Como ambos os recursos já estão alocados, o processo A e B serão bloqueados. Nessa situação eles continuarão indefinidamente, pois nenhum processo terá como se finalizar e, por fim, liberar o recurso já alocado. Esses processos geram um *deadlock*.

Uma ilustração de *deadlock* bem interessante, trazida no livro de *Silberschatz*, seria a seguinte: “Quando dois trens se aproximarem um do outro em um cruzamento, ambos deverão parar completamente e nenhum deles partirá novamente até que o outro o tenha feito”.

É importante destacar, que os impasses podem ser gerados tanto por recursos de *hardware* quanto por recursos de *software*. Um recurso de *software* seria um impasse gerado no acesso a registros

de um banco de dados. Sendo assim, trataremos o termo recurso de uma forma geral.

Os recursos podem ser divididos em dois tipos:

- **Recursos preemptíveis:** trata-se de recursos que podem ser retirados de um determinado processo sem gerar problema algum. Por exemplo, a memória.
- **Recursos não-preemptíveis:** trata-se de recursos que não podem ser retirados de um determinado processo sem causar problemas. Por exemplo, é inviável retirar a impressora de um determinado processo que começou a imprimir sua saída.

4.1. Condições para um Impasse

Coffman demonstrou em 1971 que existe quatro condições para que haja um impasse:

- Condição de exclusão mútua: somente um processo de cada vez pode acessar um recurso. Caso contrário, o recurso estará disponível.
- Condição de posse e espera: um processo deve estar de posse de um recurso e solicitando novos recursos.
- Condição de não preempção: os recursos concedidos aos processos não podem ser retirados deles.
- Condição de espera circular: deve haver uma cadeia circular de dois ou mais processos, cada um dos quais está esperando um recurso já segurado pelo próximo membro da cadeia e, assim por diante.

4.2. Modelagem de Impasse através de Grafos

Os **deadlocks** ou impasses podem ser modelados através de **grafos dirigidos** ou **grafos de alocação de recursos**. Esses grafos podem ter dois tipos de nós: processos e recursos. Os nós de processos são representados através de círculos e os nós de recursos através de quadrados, como mostrado na Figura 18.

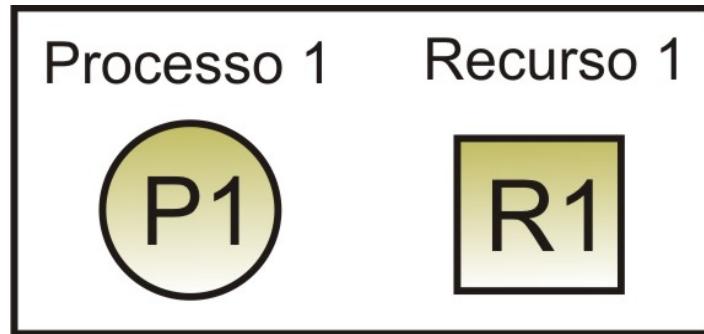


Figura 18: Nós de processos e recursos na modelagem de *deadlocks*.

Adiante, um arco de um processo apontando para um recurso significa que o processo está bloqueado esperando esse recurso. Já um arco de um recurso para um processo significa que este recurso está alocado para o processo. A Figura 19 ilustra estes dois arcos, respectivamente.

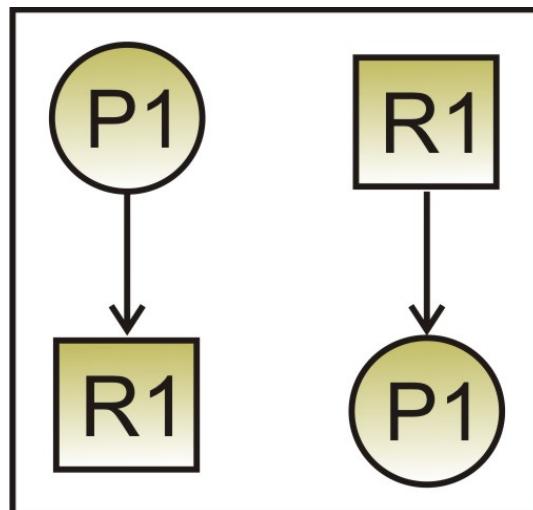


Figura 19: Arco de Processo para Recurso e de Recurso para Processo.

Definido a modelagem do grafo, podemos mostrar que se o gráfico não contiver um ciclo fechado, nenhum processo estará em uma situação de impasse. Porém, se o ciclo fechado for formado, poderemos ter um *deadlock*. A Figura 20(a) mostra um grafo sem deadlock e a Figura 20(b) mostra uma situação de impasse.

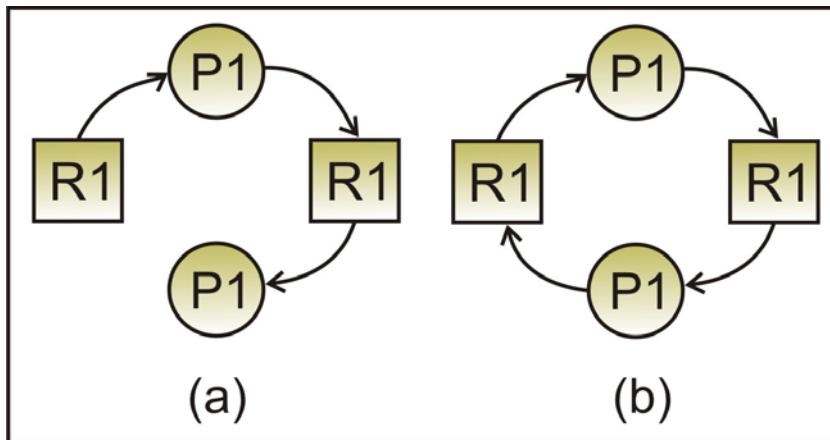


Figura 20: Modelagem através de Grafos. (a) Situação onde não está caracterizado *deadlock*. (b) situação onde se caracteriza um *deadlock*.

4.3. Métodos de Lidar com *Deadlocks*

Existem várias formas de lidar com *deadlocks*. Em geral, temos quatro métodos:

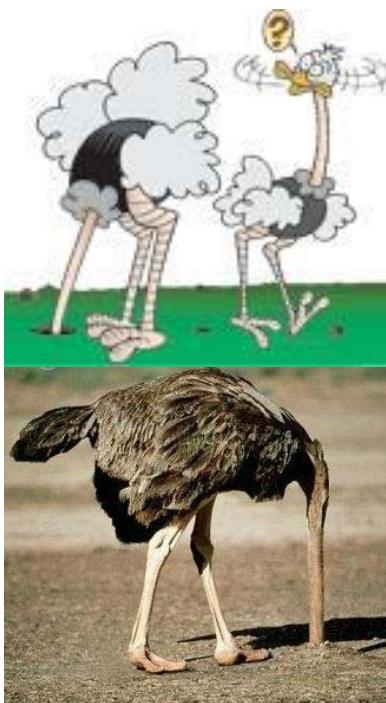
- Ignorar completamente o problema.
- Detectar e recuperar uma situação de *deadlock*.
- Prevenir um *deadlock* através da negação das quatro condições de *Coffman*.
- Impedimento de um *deadlock* através de uma alocação dinâmica cuidadosa de recursos.

Trataremos esses métodos nos sub-tópicos a seguir.

4.3.1. Algoritmo do Avestruz

A forma mais simples de lidar com *deadlock* é chamada de **Algoritmo do Avestruz**. Reza a lenda, que o avestruz, em um problema qualquer, enfia sua cabeça na areia e nem se preocupa com o que está acontecendo, ou seja, finge que não há nenhum problema.

Esta solução é utilizada em muitos Sistemas Operacionais, inclusive UNIX. Assim, quando acontece uma situação de impasse, o SO simplesmente ignora o problema, passando a responsabilidade do usuário para tomada de decisões.



O avestruz enfiar a cabeça na areia não passa de uma lenda. Confesso que eu mesmo nunca vi um avestruz com a cabeça na areia.

4.3.2. Detecção e Recuperação

Uma outra técnica para lidar com uma situação de impasse é a técnica de detecção e recuperação. Nessa técnica, o Sistema Operacional não se preocupa em prevenir um *deadlock*. Ao invés disso, o sistema fica monitorando as solicitações e liberações de recursos.

Caso um recurso seja solicitado e a nova situação gere um impasse, o Sistema Operacional tentará eliminar um processo do ciclo. Se o ciclo ainda existir, outro processo será eliminado até que o ciclo seja quebrado.

4.3.3. Prevenção de Impasses

Outra estratégia é impor situações para os processos de tal forma que um impasse seja impossível de acontecer. Essas situações seria negar pelo menos uma das condições para que ocorra um *deadlock*, modeladas por *Coffman*.

Abordaremos então cada condição de *Coffman* separadamente.

Condição de exclusão mútua

Caso nenhum recurso fosse atribuído de forma exclusiva a um único processo, negaríamos essa condição e não teríamos um impasse. Só que existe recursos impossíveis de serem atribuídos a mais de um processo, como a impressora.

Uma solução seria utilizar técnicas de *spool*, onde os processos, ao invés de acessar o recurso diretamente, acessariam um espaço de memória e gravariam suas informações. A partir desse espaço de memória, um único processo iria ler as informações e enviar para o recurso específico. Porém, nem todo recurso é possível se utilizar técnicas de *spool*.

Condição de posse e espera

Na segunda condição, se for possível fazer que um processo, que segure algum recurso, não solicite outros recursos, eliminariamós esta condição e, naturalmente, uma possível situação de impasse.

Uma solução seria fazer que, um processo solicite todos os seus recursos antes de iniciar a execução. Caso os recursos estejam disponíveis, ele executa. Caso contrário, ele precisa esperar. O problema para esta alternativa é que, na maioria dos casos, os processos não sabem *a priori* quais recursos irá precisar.

Condição de não preempção

Atacar a terceira condição de *Coffman*, sem sombra de dúvidas, é uma das soluções menos promissora, pois certos recursos são, intimamente, não preemptíveis. Mais uma vez, tomemos como exemplo a impressora. Tomar a impressora de um processo no meio da impressão é uma atitude um tanto sem lógica.

Condição de espera circular

A última condição diz que, para se ter um *deadlock*, é formado uma cadeia circular de dois ou mais processos, cada um dos quais está esperando um recurso já segurado pelo próximo membro da cadeia e, assim por diante.

Consideraremos uma possibilidade: enumerar globalmente todos os recursos disponíveis. Assim, a regra seria que processos podem solicitar recursos sempre que desejarem, mas todas as solicitações devem ser feitas em ordem numérica.

Com essa regra, nunca seria formado uma situação circular. Considere a situação de um processo 1, com um recurso 1 alocado e um processo 2 com um recurso 2 alocado. Se o processo 1 solicitar o recurso 2 e o processo 2 solicitar o recurso 1 formaríamós um ciclo e teríamos uma situação de impasse. Porém, com essa regra, o processo 2 (com recurso 2 alocado) não tem permissão para solicitar o recurso 1. Assim, o impasse é impossível. A Figura 21 ilustra essa solução.

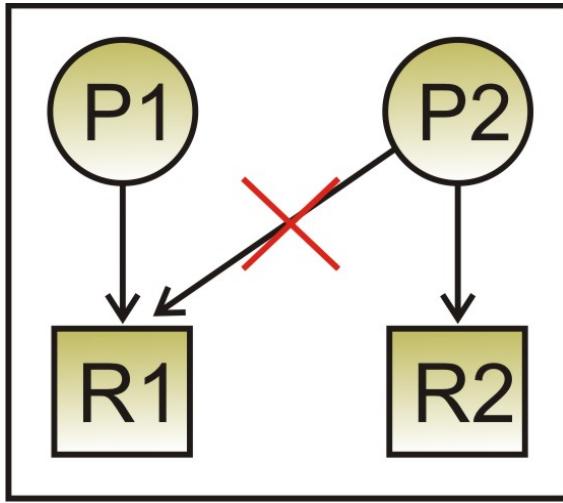


Figura 21: Recursos ordenados numericamente

4.3.4. Impedimento de Impasses

O sistema pode evitar que um impasse ocorra mediante a alocação cuidadosa dos recursos disponíveis. Para que isso ocorra, o sistema precisará de informações adicionais *a priori* de quais recursos um processo irá requisitar e usar durante sua execução.

Analisaremos o algoritmo do banqueiro para um recurso e para múltiplos recursos, que são utilizados para avaliar se uma alocação de recursos é considerada segura ou não.

Algoritmo do Banqueiro para um Único Recurso

O algoritmo do banqueiro foi proposto por *Dijkstra* (1965). Este algoritmo é baseado na mesma idéia de um banqueiro de uma pequena cidade para garantir ou não crédito a seus clientes. A idéia é que um banco jamais possa alocar todo seu dinheiro disponível em caixa de tal modo que não possa mais atender às necessidades de todos os seus clientes.

Esta analogia pode ser utilizada pelo Sistema Operacional, considerando que os clientes são os processos, os recursos são as linhas de crédito e o banqueiro trata-se do Sistema Operacional em

si. Um processo ao entrar no sistema, declara a quantidade máxima de recursos que ele necessitará para executar. Esta quantidade não pode extrapolar a quantidade deste recurso disponível no sistema.

A Figura 22 ilustra quatro processos no sistema, com a quantidade de recursos alocados e a quantidade máxima que cada um necessita para poder executar, além da disponibilidade desse recurso no sistema. Essa listagem apresentada na Figura 22 é chamada de um **estado**.

Recurso		
Processo	Alocado	Máximo
A	0	6
B	0	5
C	0	4
D	0	7

Disponíveis: 10

Figura 22: Estado inicial contendo quatro processos e a quantidade máxima de recursos necessários.

Quando um determinado processo solicitar uma quantidade dos recursos disponíveis, o sistema deverá determinar se esta alocação deixará o sistema em um **estado seguro**. Assim, considera-se um estado seguro se existir uma sequência de outros estados que leva todos os processos a solicitarem seus recursos máximos.

Nesse caso, o estado representado pela Figura 23 é considerado um **estado seguro**, pois o processo C ainda terá condição de alocar a sua quantidade de recursos máxima e, naturalmente, executar suas atividades, finalizar e liberar os

recursos para serem utilizados pelos outros processos. Dessa forma, com esse estado não teremos uma situação de *deadlock*.

Recurso		
Processo	Alocado	Máximo
A	1	6
B	1	5
C	2	4
D	4	7

Disponíveis: 2

Figura 23: Estado considerado seguro no Algoritmo do Banqueiro para Único Recurso.

Porém, se o sistema, ao invés de disponibilizar os dois recursos livres no sistema para o processo C, disponibilizasse um recurso para o processo B, o estado gerado seria um estado não seguro, pois com apenas um recurso disponível, nenhum dos processos do conjunto iria conseguir finalizar e, consequentemente, teríamos uma situação de impasse. A Figura 24 representa este estado não seguro.

Recurso		
Processo	Alocado	Máximo
A	1	6
B	2	5
C	2	4
D	4	7

Disponíveis: 1

Figura 24: Estado considerado não seguro no Algoritmo do Banqueiro para Único Recurso.

Algoritmo do Banqueiro para Múltiplos Recursos

O algoritmo do banqueiro pode ser generalizado para vários processos e várias classes de recursos, cada um com uma quantidade diversificada.

A modelagem desse algoritmo é feito através de duas matrizes: a primeira contendo os recursos alocados aos processos e a segunda contendo a quantidade ainda necessária de cada recurso por processo. Temos, também, três vetores: um **vetor E** que mostra a quantidade de recursos existentes no sistema, um **vetor A** que mostra a quantidade de recursos já alocados e um **vetor L** que mostra a quantidade de recursos livres. A Figura 26 mostra esta modelagem

Proc.	R1	R2	R3	R4	Proc.	R1	R2	R3	R4
A	0	0	0	0	A	4	1	1	1
B	0	0	0	0	B	0	2	1	2
C	0	0	0	0	C	4	2	1	0
D	0	0	0	0	D	1	1	1	1
E	0	0	0	0	E	2	1	1	0

Recursos Alocados Recursos ainda necessários
Vetor E = (6 3 4 2)
Vetor A = (0 0 0 0)
Vetor L = (6 3 4 2)

Figura 25: Modelagem do Algoritmo do Banqueiro para Múltiplos Recursos.

Note a partir da Figura 25, que, por exemplo, o processo A necessita de 4 recursos R1, 1 recurso R2, 1 recurso R3 e 1 recurso R4 para executar. Note também que o sistema contém 6 recursos R1, 3 recursos R2, 4 recursos R3 e 2 recursos R4.

O algoritmo do banqueiro para Múltiplos recursos funciona da seguinte forma:

- Procurar uma linha da matriz de recursos ainda necessários, no qual todas as posições sejam menores ou iguais ao vetor de recursos livres. Se não existir essa linha, nenhum recurso conseguirá finalizar e teremos, enfim, um impasse.
- Caso um processo execute e se finalize, os recursos por ele alocados serão disponibilizados e poderão ser utilizados por outro processo.

No caso de vários processos (linhas da matriz) tiverem condições de se finalizarem, independente de qual será escolhido, o algoritmo resultará em um estado seguro.

A Figura 26 representa um estado seguro. Note, por exemplo, que o processo D, que já possui alocado um recurso R1, um recurso R2 e um recurso R4, precisa apenas de um recurso R3. Se verificarmos o vetor de recursos livres, podemos verificar que ainda existem dois recursos R3 livres, ou seja, a linha (0 0 1 0) é menor que o vetor (1 0 2 0). Assim, este estado pode ser considerado seguro.

Proc.	R1	R2	R3	R4	Proc.	R1	R2	R3	R4
A	3	0	1	1	A	1	1	0	0
B	0	1	0	0	B	0	1	1	2
C	1	1	1	0	C	3	1	0	0
D	1	1	0	1	D	0	0	1	0
E	0	0	0	0	E	2	1	1	0

Recursos Alocados Recursos ainda necessários

Vetor E = (6 3 4 2)
Vetor A = (5 3 2 2)
Vetor L = (1 0 2 0)

Figura 26: Estado considerado seguro no algoritmo do Banqueiro para Múltiplos Recursos.

Entretanto, se fosse atribuído um recurso R3 para os processos B e E, o novo estado gerado não seria seguro, pois não existiria uma linha menor que o vetor de recursos livre. Analise através da Figura 27 esta situação impasse.

Proc.	R1	R2	R3	R4	Proc.	R1	R2	R3	R4
A	3	0	1	1	A	1	1	0	0
B	0	1	1	0	B	0	1	0	2
C	1	1	1	0	C	3	1	0	0
D	1	1	0	1	D	0	0	1	0
E	0	0	1	0	E	2	1	0	0

Recursos Alocados

Vetor E = (6 3 4 2)

Vetor A = (5 3 4 2)

Vetor L = (1 0 0 0)

Recursos ainda necessários

Figura 27: Estado considerado não seguro no algoritmo do Banqueiro para Múltiplos Recursos.

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI

www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB

www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED

www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED

www.abed.org.br

Curso de Sistemas Operacionais – DCA/UFRN

Prof. Dr. Luiz Affonso Henderson Guedes de Oliveira

<http://www.dca.ufrn.br/~affonso/DCA0108/curso.html>

Home Page do Prof. Dr. Rômulo Silva de Oliveira

<http://www.das.ufsc.br/~romulo/>

Home Page do Autor Andrew S. Tanenbaum

<http://www.cs.vu.nl/~ast/>

Simulador de Ensino para Sistemas Operacionais

<http://www.training.com.br/sosim/>

REFERÊNCIAS BIBLIOGRÁFICAS

BACON, J. e HARRIS, T. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

DEITELL, H. M. & DEITEL, P. J. & CHOHNES, D. R. *Sistemas Operacionais*. São Paulo, Ed. Prentice Hall, 2005.

MACHADO, F. e MAIA, L. *Arquitetura de Sistemas Operacionais*. 4^a Edição, LTC, 2007

OLIVEIRA, R. S. e CARISSIMI, A. S. e TOSCANI, S. S. *Sistemas Operacionais*. 3^a ed, volume 11, Ed. Bookman, 2008.

SHAW, A. C. *Sistemas e software de tempo real*. Porto Alegre, Ed. Bookman, 2003.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Fundamentos de Sistemas Operacionais*. 6^a Edição, Ed. LTC.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Sistemas Operacionais com Java*. 7^a Edição, Rio de Janeiro, Ed. Elsevier, 2008.

TANENBAUM, A. S. & WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*. 3^a Edição, Porto Alegre, Ed. Bookman, 2008.

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



Vimos no Capítulo II que na multiprogramação, os processos alternam suas execuções na CPU, passando a impressão de que eles estão sendo executados ao mesmo tempo (pseudoparalelismo). Para que alternância entre os processos para usar o processador seja rápido, esses processos precisam estar em memória.

Porém, a memória é um recurso escasso e manter uma quantidade cada vez maior de processos carregados em memória exige cada vez do Sistema Operacional capacidade de gerenciamento correto e eficaz. Sendo assim, o bom método de gerenciamento de memória influencia diretamente no desempenho de um Sistema Computacional.

A parte do Sistema Operacional, responsável por gerenciar a memória, controlar as partes de memória que estão e que não estão em uso e alocar e desalocar memória para os processos, é conhecida como **gerenciador de memória**.

Nesta unidade trataremos dos principais métodos de gerenciamento de memória, dos mais simples, através de alocação contígua, até os mais complexos, como exemplo o método de memória virtual.

SUMÁRIO

UNIDADE IV – GERENCIAMENTO DE MEMÓRIA

1. INTRODUÇÃO
 2. GERENCIAMENTO BÁSICO DE MEMÓRIA
 3. GERENCIA DE MEMÓRIA PARA MULTIPROGRAMAÇÃO
 - 3.1. Alocação com Partições Variáveis
 - 3.1.1. Gerenciamento de Memória com Mapa de Bits
 - 3.1.2. Gerenciamento de Memória com Listas Encadeadas
 4. MEMÓRIA VIRTUAL
 - 4.1. Paginação
 - 4.2. TLB – *Translation Lookside Buffers*
 5. ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINAS
 - 5.1. Algoritmo de Substituição de Página Ótimo
 - 5.2. Algoritmo de Substituição de Página Não Recentemente Utilizada
 - 5.3. Algoritmo de Substituição de Página FIFO
 - 5.4. Algoritmo de Substituição de Página de Segunda Chance
 - 5.5. Algoritmo de Substituição de Página do Relógio
 - 5.6. Algoritmo de Substituição de Página menos Recentemente Utilizada
- WEBLIOGRAFIA
- REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE IV

GERENCIAMENTO DE MEMÓRIA

1. INTRODUÇÃO

A memória é um recurso fundamental e de extrema importância para a operação de qualquer Sistema Computacional, seja os mais antigos até os mais modernos. De fato a memória trata-se de uma grande região de armazenamento formada por bytes ou palavras, cada uma contendo seu próprio endereço.

Os Sistemas Computacionais mais antigos possuíam uma quantidade muito escassa de memória. Atualmente, a memória já é um recurso bem mais acessível e muito mais disponível que antigamente. É comum, por exemplo, uma pessoa ir até uma loja de informática e comprar um computador com uma quantidade de memória RAM bem razoável (2GB).

Porém, mesmo com o ambiente atual sendo muito mais favorável que antigamente, o gerenciamento da memória é um dos fatores mais importantes e complexos de qualquer projeto de Sistemas Operacionais, pois influencia diretamente no bom ou mau desempenho de um Sistema Computacional.

Tanenbaum aponta uma situação que seria a mais favorável: imagine que todo programador tivesse à sua disposição uma memória muito rápida e infinitamente grande que também fosse não-volátil (ou seja, não perde seus dados com a ausência de energia) e, por aquele preço camarada. Essa situação chega a ser bastante utópico. De fato, essas características são, um tanto quanto, contraditórias. Por exemplo, quanto mais rápido e maior a memória, com certeza, muito maior o preço.

A maioria dos computadores tem uma **hierarquia de memória**, que trata-se de uma pirâmide, no qual em direção ao topo

estão as memórias mais rápidas, porém, mais caras. A Figura 28 esboça essa pirâmide.

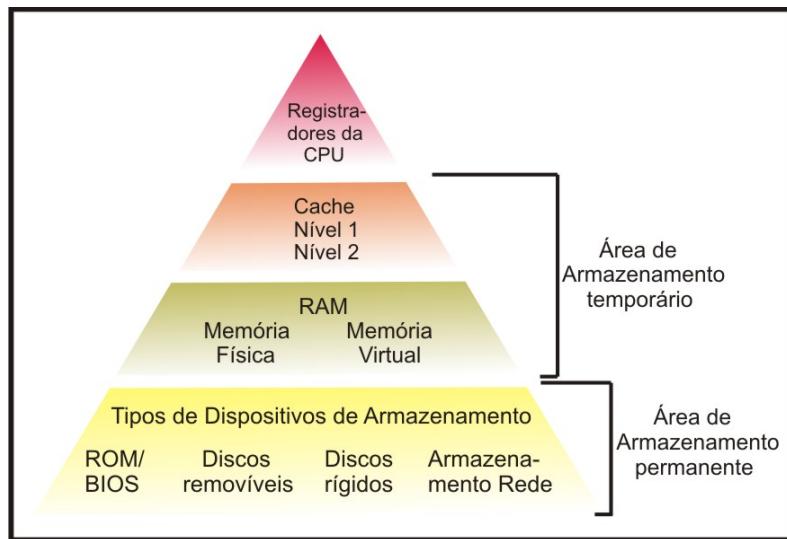


Figura 28: Hierarquia de Memória

O Sistema Operacional tem como principal função gerenciar a hierarquia de memória. A parte do SO responsável por essa função é chamada de **gerenciamento de memória**. Entre as funções do gerenciador de memória está em controlar as partes de memória utilizada e as partes não utilizadas, alocar (disponibilizar) memória para os processos, desalocar (retirar) memória de processos e gerenciar a troca entre memória principal (primária) e memória secundária.

Segundo Tanenbaum, basicamente, os métodos de gerenciamento são divididos em duas grandes classes: os métodos que fazem troca ou paginação e os que não fazem. Iremos estudar alguns desses métodos adiantes.

2. GERENCIAMENTO BÁSICO DE MEMÓRIA

O esquema mais simples de gerenciamento de memória foi implementado nos primeiros Sistemas Operacionais, porém ainda está presente em alguns sistemas monoprogramáveis. Este esquema é chamado por alguns autores como alocação contígua.

Basicamente, a memória principal disponível é dividida entre o Sistema Operacional e o programa em execução. Como este esquema de gerenciamento é utilizado em sistemas monoprogramáveis, temos apenas um processo em execução por vez. A Figura 29 ilustra esse esquema de gerenciamento de memória principal.

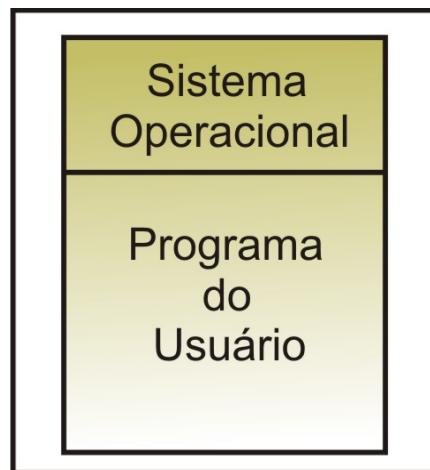


Figura 29: Esquema Básico de gerenciamento de memória

Neste tipo de gerenciamento, o usuário tem acesso a toda memória principal, inclusive o espaço do Sistema Operacional. É possível proteger a área de memória do Sistema Operacional utilizando registradores delimitadores de área de usuário e do SO.

Assim, os programas estão limitados ao tamanho de memória principal disponível. Uma técnica conhecida como **overlay**, permite que um programa seja dividido em módulos, de forma que seja possível a execução independente de cada módulo, utilizando uma mesma área de memória.

3. GERENCIA DE MEMÓRIA PARA MULTIPROGRAMAÇÃO

Atualmente, os sistemas multiprogramáveis são amplamente mais utilizados do que os sistemas monoprogramáveis, devido a maior eficiência do uso do processador, pois permitem que múltiplos processos executem simultaneamente.

A forma mais simples de gerenciar memória em sistemas multiprogramáveis é dividindo a memória principal em partições estáticas e com tamanhos definidos, estabelecidas na fase de inicialização do sistema. Esse tipo de gerência é chamado por alguns autores por **alocação particionada estática** ou **alocação fixa**.

Quando um processo chega, ele é colocado em uma fila de entrada da partição menor capaz de armazená-lo. Como o esquema prevê que a partição é fixa, se o processo não ocupar o espaço total da sua partição, o resto de espaço é desperdiçado. A Figura 30 mostra esse esquema de gerenciamento de memória.

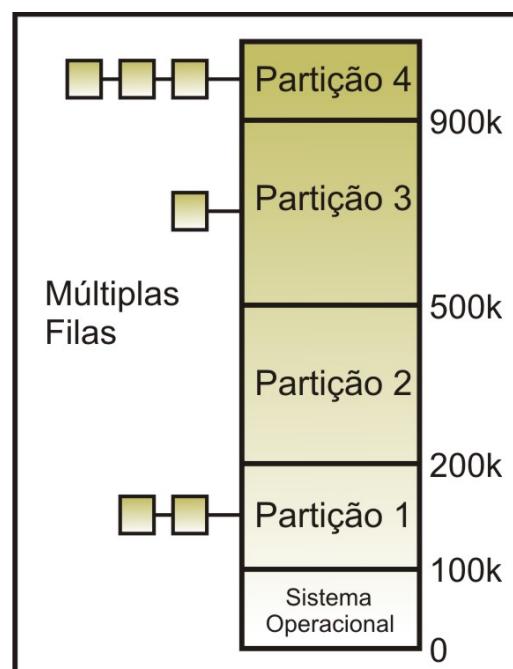


Figura 30: Esquema de gerencia de memória para multiprogramação com Partições Fixas.

Assim, os programas eram carregados em uma partição específica. Em algumas situações, é possível que uma determinada fila exista uma quantidade grande de processos esperando pela partição enquanto que em outras filas de partição não existe nenhum processo. Isso era uma grande desvantagem para o sistema. Por exemplo, na Figura 30 podemos verificar que a partição 4 possui três processos em espera, enquanto que na partição 2 não existe nenhum processo. Essa partição poderia, assim, ser utilizado por um desses processos na fila.

Dessa forma, uma solução para tentar contornar esse problema é implementar este método com uma única fila. Assim, quando uma determinada partição estivesse livre, o processo mais próximo do início da fila que melhor se ajusta à essa partição poderia ser carregado nela. Este método é mostrado na Figura 31.

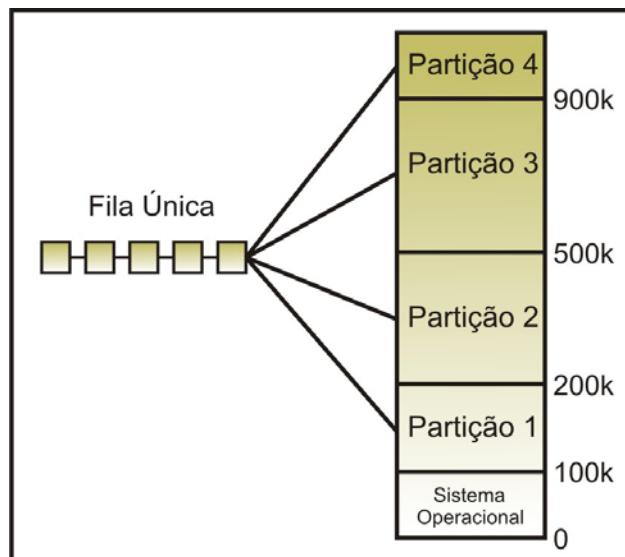


Figura 31: Método de partições fixas com única fila.

Este método de gerência de memória baseado em partições fixas, de uma forma ou de outra, gera um grande desperdício de memória, posto que, um processo ao ser carregado em uma partição, se ele não ocupa todo o seu espaço, o restante não poderá ser utilizado por nenhum outro processo.

Para evitar esse desperdício, foi desenvolvido um esquema de gerenciamento e alocação de memória dinamicamente, dependendo da necessidade do processo. Este esquema é conhecido como **alocação com partições variáveis**. Estudaremos este método no próximo tópico.

3.1. Alociação com Partições Variáveis

Quando o método de alocação de partições variáveis é utilizado, o tamanho das partições de memória é ajustado dinamicamente à medida que os processos vão chegando.

Nesse tipo de esquema, o processo utilizará um espaço de memória necessário, tornando esse espaço sua partição. Isto resolve o problema da fragmentação, pois as partições passarão a ser definidas a partir da alocação dos processos na memória.

O método é bem simples. Observe através da Figura 32, que um processo A ao chegar, é carregado na memória e o seu tamanho define do tamanho da sua partição. O processo B e C, de forma análoga, são carregados no espaço livre, ainda disponível. Ao chegar o processo D, não existe espaço livre contínuo suficiente para ele ser carregado.

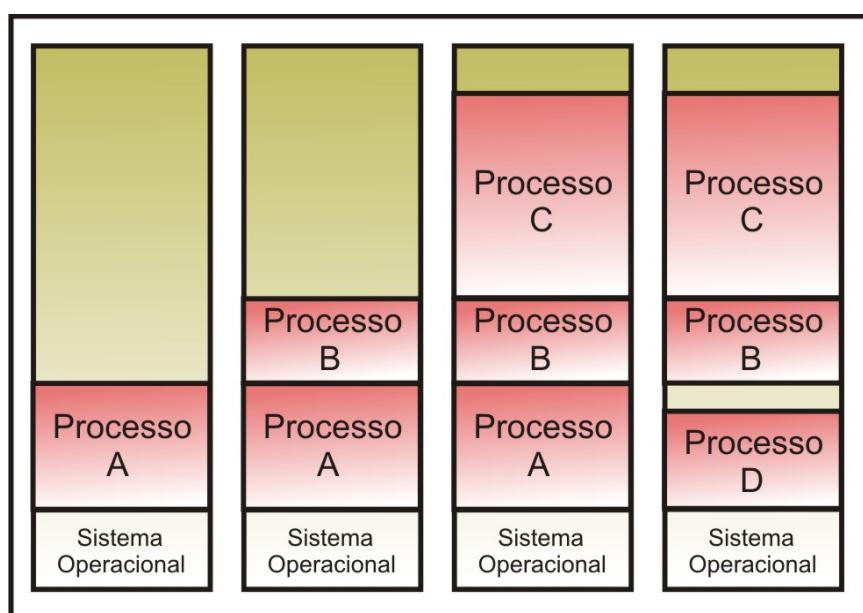


Figura 32: Esquema de Alocação com Partições Variáveis.

Note que, para inserir o novo processo D na memória foi preciso retirar o processo A, a fim de o espaço de memória suficiente seja disponibilizado para o novo processo. Assim, o processo A é atualizado em disco e o processo D é carregado em memória principal, podendo executar suas atividades. Este processo de retirar um processo de memória, atualizar em disco e colocar outro no lugar é chamado, segundo Tanenbaum, por método de **Troca**. A troca consiste em trazer um processo inteiro, executa-lo temporariamente e, então, devolve-lo ao disco. Isto acontece quando a memória principal disponível é insuficiente para manter todos os nossos processos carregados (situação que seria a ótima). Existe outra estratégia, que permite que apenas parte do programa fique em memória principal. Esta estratégia, bem mais complexa que a troca, é conhecida como **Memória Virtual**. Veremos esta estratégia um pouco mais a frente.

Podemos notar através da Figura 32, que após alocar e desalocar vários processos, a memória pode se particionar (espaço entre os processos D e B, por exemplo). Chamamos isso de fragmentação externa. Veja que essa fragmentação é diferente da gerada na alocação estática. Na alocação estática, a fragmentação é gerada pelo não uso total da partição pelo processo (fragmentação interna).

Assim, quando a troca cria várias lacunas na memória, é possível juntar todas essas lacunas em um grande espaço, movimentando todos os processos para baixo o máximo possível. Essa técnica é conhecida como compactação de memória e exige muito tempo de CPU.

O método de alocação dinâmica da partição é bem mais flexível que o método de alocação estática. Porém, essa flexibilidade também complica mais a tarefa de alocar e desalocar a memória, assim como a monitoração da memória utilizada. O Sistema Operacional tem a função de gerenciar essa memória. De modo geral, há duas maneiras de monitorar o uso da memória: através do

mapa de bits e listas encadeadas. Veremos essas duas estratégias nos sub-tópicos a seguir:

3.1.1. Gerenciamento de Memória com Mapa de Bits

Com o mapa de bits, a memória é dividida em unidades de alocação e para cada unidade é associado um *bit* no mapa. Se este *bit* estiver com valor 0 implica que esta unidade de alocação está vazia. Se este *bit* estiver com valor 1 implica que esta unidade de alocação está preenchida por algum processo.

Através da Figura 33 é mais fácil compreender como essa estratégia de gerenciamento funciona. Considere a memória, partitionada em XX unidades de alocação, com processos e espaços vazios (**lacunas**). O mapa de *bits* dessa memória será configurado como mostrado na figura.

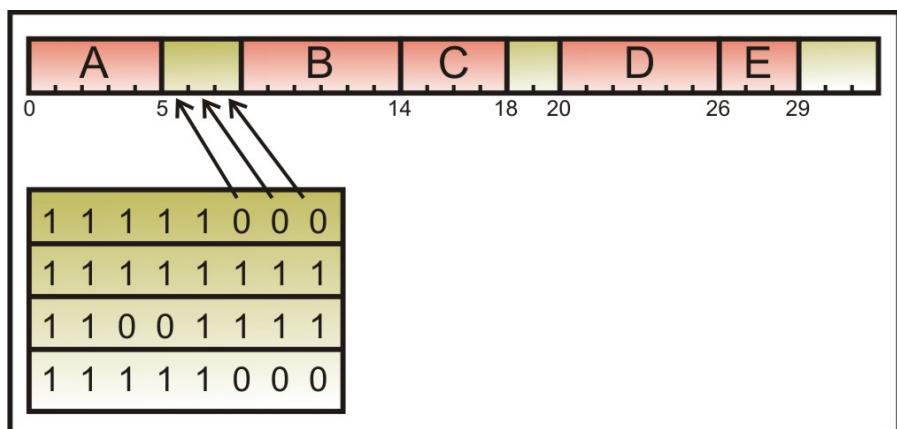


Figura 33: Memória dividida em unidade de alocação e o Mapa de *bits* correspondente.

O ponto crucial do mapa de *bits* é a definição do tamanho da unidade de alocação. Se ele for muito pequeno, o mapa de *bits* poderá ser muito grande. Se a unidade for muito grande, o mapa de *bits* será menor, mas uma quantia de memória poderá ser desperdiçada na última unidade se o tamanho do processo não for um múltiplo exato da unidade de alocação.

3.1.2. Gerenciamento de Memória com Listas Encadeadas

Outra estratégia é manter uma lista encadeada dos segmentos de memória alocados e livres, onde um segmento pode ser um processo ou lacuna entre dois processos.

Assim, cada nó da lista seria formado por uma entrada especificando se é um processo (P) ou uma lacuna (L), o endereço de onde se inicia, a quantidade de unidades de alocação e um ponteiro para a próxima entrada. A Figura 34 ilustra a mesma memória gerenciada através de listas encadeadas.

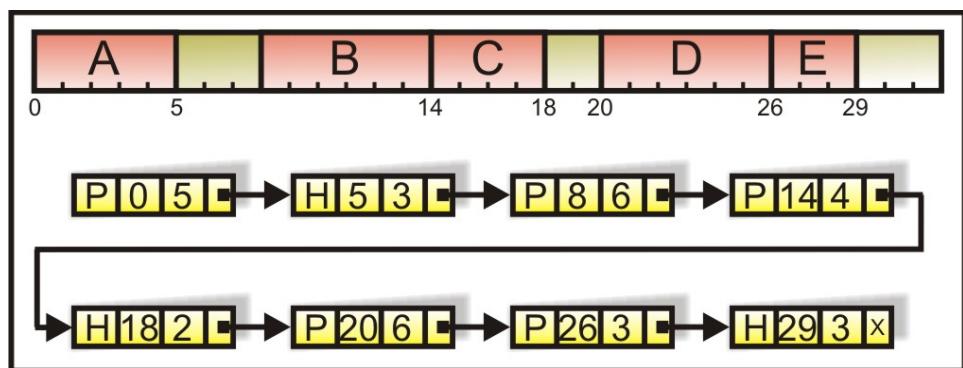


Figura 34: Memória dividida em unidade de alocação, monitorada através de Lista Encadeada.

4. MEMÓRIA VIRTUAL

Como foi dito nos tópicos anteriores, os programas estão limitados ao tamanho de memória principal disponível. É possível utilizar a técnica *overlay*, que permite que um programa seja dividido em módulos, de forma que seja possível a execução independente de cada módulo, utilizando uma mesma área de memória.

Trabalhar com *overlays* joga a responsabilidade para o programador de dividir o seu programa, o que trata-se de uma tarefa complexa e que exigia tempo do programador. O ideal seria que a divisão do programa fosse feito pelo sistema.

De fato, em 1961, *Fotheringham* desenvolveu um método conhecido como **memória virtual**. O principal objetivo desta técnica é estender a memória principal através de memória secundária, dando impressão ao usuário que ele tem a disposição uma quantidade de memória maior do que a memória real disponível. Em outras palavras, para facilitar o entendimento, a técnica de memória virtual disponibiliza para o usuário uma memória de trabalho maior do que a memória principal (RAM). Esse espaço de memória adicional implementado em disco é conhecido como **memória de swap**. A Figura 35 mostra um desenho ilustrativo desta técnica.

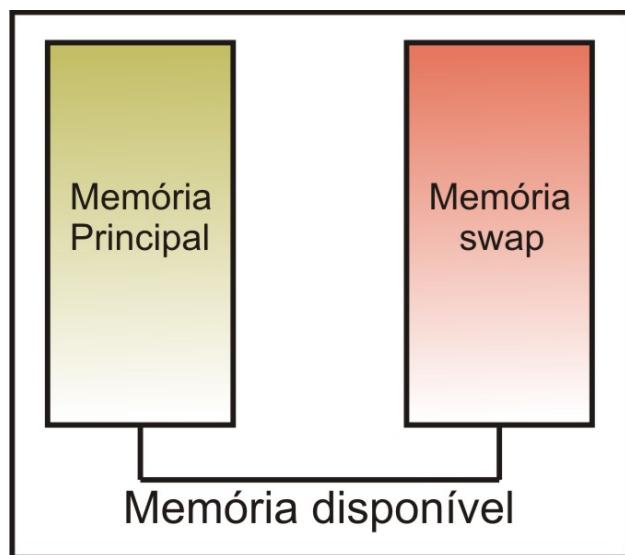


Figura 35: Ilustração do método de memória virtual.

Assim, o Sistema Operacional deve manter parte do programa que está em uso na memória principal e parte dele na memória swap. Na multiprogramação, partes de vários processos podem ser mantidas em memória principal e partes em memória swap.

Existem algumas estratégias para implementar o método de memória virtual. A estratégia mais conhecida é chamada de paginação. Veremos essa estratégia no tópico a seguir.

4.1. Paginação

Segundo *Silberstchaz*, a paginação é um esquema de gerenciamento de memória em que o espaço de endereçamento físico (endereços de memória principal) de um processo não é contíguo. A paginação, assim, evita o problema da fragmentação externa gerado pela alocação de memória dinâmica.

Os programas são capazes de gerar endereços, chamados de **endereços virtuais**. Esse conjunto de endereços forma o que chamamos de **espaço de endereçamento virtual**. Em sistemas que não utilizam a técnica de memória virtual, o endereço virtual equivale ao endereço físico, especificando exatamente onde o programa será armazenado na memória principal.

Em computadores que utilizam a técnica de memória virtual, os endereços virtuais não vão diretamente para o barramento de memória. Primeiramente, ele passa por uma unidade chamada de **Unidade de Gerenciamento de Memória (MMU – Memory Management Unit)**. A MMU tem como função mapear um endereço virtual para um endereço lógico.

Adiante, o espaço de endereçamento virtual é dividido em unidades chamadas de **páginas**. A memória principal também é dividida em unidades, do mesmo tamanho das páginas, chamadas de **molduras de página** (quadro). Assim, num sistema que contém páginas de 4k, as molduras também serão de 4k. Considere a Figura 36, com o espaço de endereço virtual formado de 64k e a memória física de 32k, com páginas de molduras de 4k.

Note através da figura, que temos 16 páginas virtuais e 8 molduras de páginas. Quando um programa tenta acessar um endereço virtual, este é passado para a MMU, que vai analisar o mapeamento deste endereço e descobrir a qual moldura pertence. A MMU mapeia este endereço virtual para o endereço físico, que, por fim, é colocado no barramento de memória.

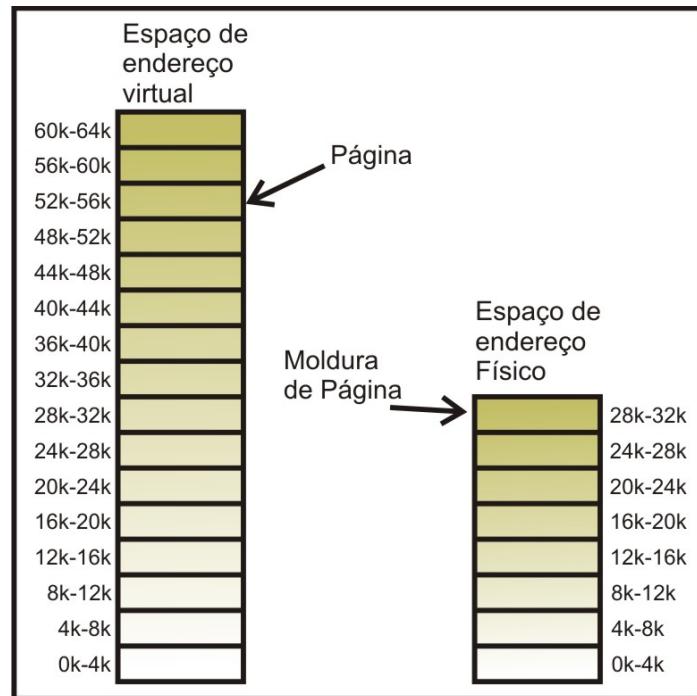


Figura 36: Técnica de Paginação

Vejamos um exemplo. Considere o mapeamento apresentado na Figura 37.

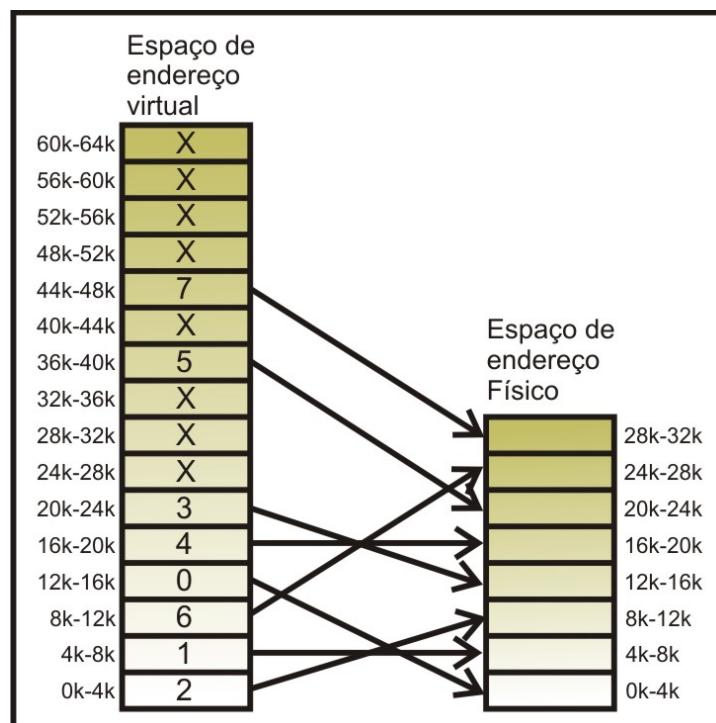


Figura 37: Mapeamento da memória virtual.

Com esse mapeamento, o endereço virtual 0, que encontra-se na página 0 (0k-4k) é mapeado pela MMU para a moldura 2 (8k-12k). De forma análoga, o endereço virtual 8195, pertencente à página 2 (8k-12k, correspondente a 8192-12287) é mapeado para a moldura 6 (24576 até 28671), e seu endereço físico será 24576 (endereço inicial da moldura) adicionado do deslocamento dele na página (página começa com endereço 8192, como seu endereço é 8195, ele está deslocado de 3). Assim, o endereço físico será 24576 + 3, que equivale a 24579.

Vejamos através desse exemplo, como a MMU faz esse mapeamento. O endereço virtual 8195 corresponde, em codificação binária, a 0010000000000011. Este endereço virtual é formado de 16 *bits*, no qual os quatro primeiros *bits* (0010) são utilizados para representar o número da página (0010 corresponde à página 2) e o restante (12 *bits*) é utilizado para representar o deslocamento na página. Como temos 4 *bits* para representar o número de páginas, com essa combinação, podemos gerar números de 0 a 15, ou seja, as 16 páginas no espaço de endereçamento virtual. Com os 12 *bits* de deslocamento podemos ter endereços de 0 a 4k, ou seja, o tamanho das páginas.

Já no espaço de endereçamento físico, como temos 8 molduras no exemplo apresentado, precisamos de 3 *bits* para representá-las. Com os 3 *bits* podemos representar as molduras 0 a 7. Como o tamanho de cada moldura corresponde também à 4k, precisamos, assim, de 12 *bits* para representar endereços de 0 a 4k. Assim, o endereço físico é representado por 15 *bits*.

Todo mapeamento realizado pela MMU é feito através de uma tabela conhecida como **tabela de páginas**. Cada processo possui sua tabela própria e cada página possui uma entrada nesta tabela. Cada entrada, ou seja, cada página na tabela possui um *bit* (**bit de validade**) que informa se a página está ou não mapeada na memória principal.

A Figura 38 mostra o exemplo do mapeamento do endereço virtual 8195 (**001000000000011**) para o endereço físico 24579 (**110000000000011**).

Quando um programa tenta utilizar um endereço virtual e a MMU percebe que a página no qual o endereço pertence não possui mapeamento na memória principal, ela gera uma interrupção no Sistema Operacional, conhecido como **falha de página (page fault)**. O Sistema Operacional, por sua vez, toma de conta do processador e seleciona uma das molduras, atualizando seu valor em disco. Então, ele busca a página não mapeada, carrega ela na moldura, altera a tabela e reinicia o processo interrompido. A moldura a ser retirada é escolhida baseada em algoritmos de escalonamento, chamados de **algoritmos de substituição de páginas**, que veremos mais a seguir.

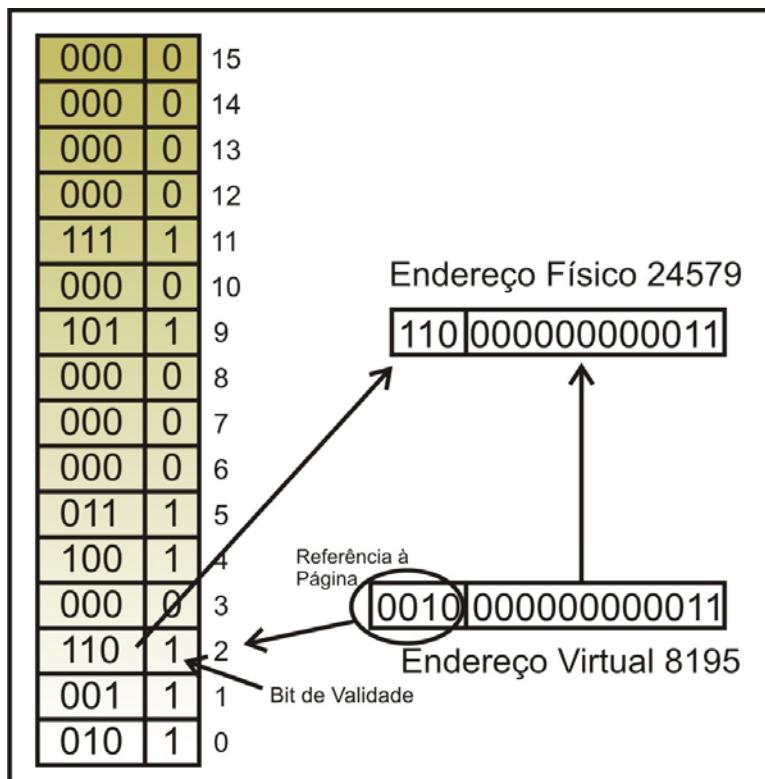


Figura 38: Mapeamento de endereço virtual para endereço físico

4.2. TLB – *Translation Lookside Buffers*

As tabelas de páginas, comumente, são mantidas em memória, o que influencia diretamente no desempenho, já que as tabelas são bastante grandes.

Segundo Tanenbaum, um comportamento que foi verificado é que programas tendem a fazer um grande número de referências a um pequeno número de páginas. Assim, somente uma pequena fração de entradas da tabela de páginas é intensamente lida.

Com isso, foi desenvolvido um dispositivo de hardware capaz de mapear endereços virtuais em endereços físicos sem passar pela tabela de páginas. Esse dispositivo é conhecido como **memória associativa** ou ***Translation Lookside Buffers*** (TLB). Comumente, ele é encontrado dentro da MMU.

5. ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINAS

Como dito anteriormente, quando ocorre uma falha de página, o Sistema Operacional deve selecionar uma das molduras para disponibilizar para a página não mapeada. Se a página a ser retirada da moldura tiver sido modificada, ela deve ser atualizada em disco. Caso contrário, a página pode ser retirada da moldura, ou seja, da memória principal, sem necessitar de atualização em memória. A página escolhida é, comumente, chamada de **página vítima**.

A escolha aleatória de uma página vítima pode influenciar diretamente o desempenho do sistema. O ideal seria retirar uma página que não é utilizada constantemente. Por exemplo, se for retirado uma página que é muito utilizada, provavelmente ela seja logo referenciada e necessite voltar para a memória principal, gerando novamente uma falha de página.

Para a escolha de qual página será retirado da memória, temos vários algoritmos de substituição de páginas, que serão descritos nos tópicos a seguir.

5.1. Algoritmo de Substituição de Página Ótimo

O algoritmo de substituição de página ótimo é o melhor algoritmo possível. A idéia básica deste algoritmo é que, quando ocorrer uma falha de página, a página a ser substituída na memória física será a que for referenciada mais tarde. Em outras palavras, se tivermos uma quantidade x de páginas carregadas na memória físicas (nas molduras), algumas páginas poderão ser utilizadas logo em seguida, porém, algumas páginas só irão ser referenciadas depois de muito tempo.

Dessa forma, se a página escolhida for a página que será referenciada mais tarde, o algoritmo estará adiando o máximo possível uma nova falha de página. Um exemplo mais prático para que possamos entender este algoritmo, seria supor que temos 10 páginas na memória. Desses 10 páginas, a página 5, por exemplo, será a primeira a ser referenciada, ou seja, a próxima. Da mesma forma, a página 0, por exemplo, será a última a ser referenciada, ou seja, do conjunto de páginas carregadas, ela será a ultima. Então, baseado no algoritmo de substituição de página ótima, a página 0 será a melhor escolha, pois estamos adiando o máximo possível uma falha de página.

É visível que este algoritmo não é realizável na prática, pois seria necessário, quem sabe, uma bola de cristal para podermos adivinhar qual seria a última página a ser referenciada. O fato de não termos como “prevê o futuro”, esse algoritmo se torna não praticável.

Porém, este algoritmo é muito utilizado para ser comparado com algoritmos realizáveis.

5.2. Algoritmo de Substituição de Página Não Recentemente Utilizada

Alguns *bits* são utilizados na tabela de páginas e são utilizados com o objetivo de facilitar a implementação de algoritmos de substituição de páginas.

O primeiro *bit* é conhecido como o *bit sujeira (dirty bit)* e indica se uma página foi alterada ou não durante a execução do processo. Toda vez que uma determinada página é gravada na memória secundária, o Sistema Operacional zera este *bit*.

O segundo *bit* é conhecido como o *bit de referência (reference bit)* e serve para indicar se uma página foi referenciada (acessada) por um processo. Quando uma página é carregada na memória, este bit é zerado.

Dessa forma, o algoritmo de substituição de página não recentemente define que, quando ocorre uma falha de página, o Sistema Operacional inspeciona todas as páginas e agrupa em quatro conjuntos, como segue:

- Conjunto 1: páginas não-referenciadas (bit de referência com valor zero) e não-modificadas (bit sujeira com valor zero).
- Conjunto 2: páginas não-referenciadas (bit de referência com valor zero) e modificadas (bit sujeira com valor 1).
- Conjunto 3: páginas referenciadas (bit de referência com valor 1) e não-modificadas (bit sujeira com valor zero).
- Conjunto 4: páginas referenciadas (bit de referência com valor 1) e modificadas (bit sujeira com valor 1).

O conjunto 4, visivelmente, contém as páginas que são menos aconselhadas de serem retiradas. O conjunto que contém páginas modificadas que não foi referenciada é uma escolha melhor do que páginas não-modificadas, mas que são constantemente referenciadas.

5.3. Algoritmo de Substituição de Página FIFO

O algoritmo de substituição de páginas FIFO (*first in first out*) define que a página que será retirada é a página que está mais tempo na memória. Em suma, trata-se de uma fila, onde o primeiro que entra é sempre o primeiro que sai.

Para exemplificar este algoritmo, consideremos a sequência de páginas solicitadas apresentada na Figura 39. Considere também

que existe apenas 3 molduras (m_1 , m_2 e m_3). Dessa forma, através do algoritmo de substituição de página FIFO temos a configuração de seleção de páginas a serem retiradas apresentadas.

Note através da Figura 39, que as primeiras páginas, pelo fato de não estarem carregadas nas molduras, também geram uma falha de página e precisam, assim, serem mapeadas. Note também que a quarta página quando é referenciada (página 3) não está mapeada. Como a página 0 foi a última a ser referenciada (mapeada), ela é a página escolhida pelo algoritmo.

Podemos verificar facilmente, que o algoritmo de substituição de página FIFO não leva em consideração se uma determinada página é muito referenciada, ou seja, caso ela seja a escolhida para sair, ela será substituída. Veja ainda na Figura 39, que a página 7 foi escolhida para sair e, logo após, ela foi novamente referenciada.

página	m_1	m_2	m_3	falha
0	0			x
2	0	2		x
1	0	2	1	x
3	3	2	1	x
5	3	5	1	x
4	3	5	4	x
6	6	5	4	x
3	6	3	4	x
7	6	3	7	x
4	4	3	7	x
7	4	3	7	
3	4	3	7	
3	4	3	7	
5	4	5	7	x
5	4	5	7	
3	4	5	3	x
1	1	5	3	x
1	1	5	3	
7	1	7	3	x

Figura 39: Algoritmo de Substituição de Página FIFO

5.4. Algoritmo de Substituição de Página de Segunda Chance

Como pudemos notar no algoritmo de substituição de página FIFO, este não leva em consideração se uma página é muito utilizada. Assim, a fim de se tentar evitar que uma página muito utilizada seja selecionada, podemos inspecionar o bit de referência antes de selecionar a página em questão.

Este algoritmo é comumente chamado de **algoritmo de substituição de página de segunda chance**. De fato, antes de se retirar uma página, é verificado se seu bit de referência tenha valor 1. Caso ele tenha valor 1, seu valor é mudado para 0 e esta página não é retirada. Note que este algoritmo é uma pequena modificação do algoritmo FIFO.

5.5. Algoritmo de Substituição de Página do Relógio

Outro algoritmo que utiliza os *bits* de referência para selecionar a próxima página a ser retirada é o **algoritmo de substituição de página do relógio**. A principal diferença para o algoritmo apresentado anteriormente está na implementação.

O algoritmo de segunda chance é implementado através de uma fila FIFO, utilizando os *bits* de referência. Já o algoritmo do relógio é implementado a partir de uma fila circular. Assim, a próxima página a ser retirada será a primeira página apontada pelo ponteiro do relógio que contiver o bit de referência 0. A Figura 40 ilustra esse algoritmo.

Note neste algoritmo, que a próxima página que será retirada não será nem as páginas 15, 16, 1 e 2, pois seus *bits* de referência tem valores 1. Assim, a próxima vítima será a página 3. A Figura 41 mostra essa configuração.

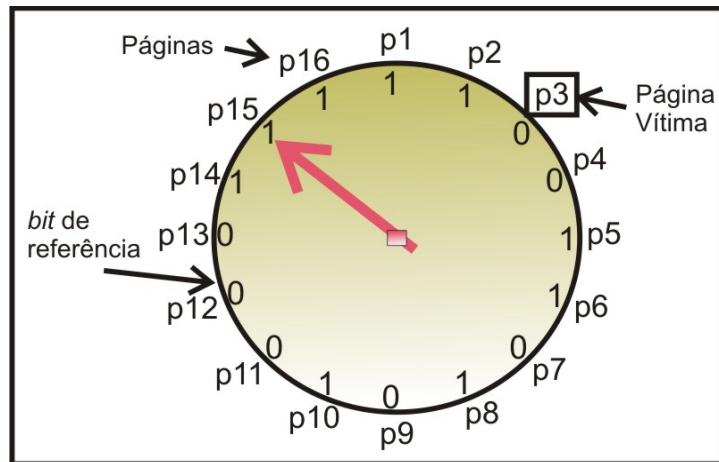


Figura 40: Algoritmo de Substituição de Página do Relógio

Observe na Figura 41 que os bits das páginas 15, 16, 1 e 2 foram zeradas e a página 17 entrou no lugar da página 3.

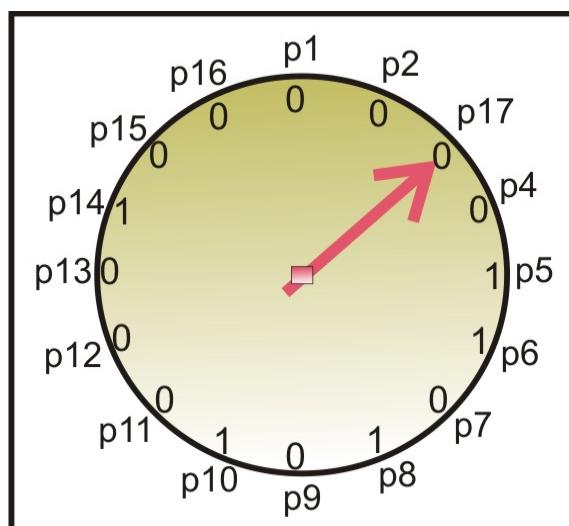


Figura 41: Seleção de página segundo o algoritmo de Substituição de Página do Relógio

5.6. Algoritmo de Substituição de Página menos Recentemente Utilizada

O algoritmo de substituição de página menos recentemente utilizada (LRU – *Least Recently Used*) trata-se do algoritmo que tem uma boa aproximação do algoritmo ótimo. O seu funcionamento é bem simples: como é impossível olhar para o futuro, ou seja, não é possível saber das páginas carregadas, qual será a última que será

referenciada, olharemos para o passado. Assim, quando acontecer uma falha de página, o algoritmo selecionará para ser retirada a página que foi a mais tempo referenciada.

A implementação do algoritmo LRU é muito custoso, segundo Tanenbaum, pois é preciso manter uma lista encadeada de todas as páginas na memória, com as páginas mais recentemente utilizadas na frente as menos recentemente utilizadas no fundo. Essa lista deve ser atualizada a cada referência de página, o que pode gerar uma grande modificação na lista. Imagine que, uma página do fim da lista (pouco referenciada) seja referenciada. É preciso, então deslocar esta página para o início da fila.

Para exemplificar este algoritmo, consideremos a mesma sequência de páginas solicitadas apresentada no algoritmo FIFO. A Figura 42 ilustra essa situação.

página	m1	m2	m3	falta
0	0			x
2	0	2		x
1	0	2	1	x
3	3	2	1	x
5	3	5	1	x
4	3	5	4	x
6	6	5	4	x
3	6	3	4	x
7	6	3	7	x
4	4	3	7	x
7	4	3	7	
3	4	3	7	
3	4	3	7	
5	5	3	7	x
5	5	3	7	
3	5	3	7	
1	5	3	1	x
1	5	3	1	
1	5	3	1	
7	7	3	1	x

Figura 42: Algoritmo de Substituição de Página menos Recentemente Utilizada

Observe, que de forma análoga ao algoritmo FIFO, as primeiras páginas, pelo fato de não estarem carregadas nas molduras, também geram uma falha de página e precisam, assim, serem mapeadas. Veja que a página 3, que no algoritmo FIFO seria a página retirada para ser substituída pela página 5, pelo fato dela ter sido referenciada, o algoritmo LRU já escolheu outra página para ser retirada, no caso a página 4, que no momento era a menos referenciada.

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI

www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB

www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED

www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED

www.abed.org.br

Curso de Sistemas Operacionais – DCA/UFRN

Prof. Dr. Luiz Affonso Henderson Guedes de Oliveira

<http://www.dca.ufrn.br/~affonso/DCA0108/curso.html>

Home Page do Prof. Dr. Rômulo Silva de Oliveira

<http://www.das.ufsc.br/~romulo/>

Home Page do Autor Andrew S. Tanenbaum

<http://www.cs.vu.nl/~ast/>

Simulador de Ensino para Sistemas Operacionais

<http://www.training.com.br/sosim/>

REFERÊNCIAS BIBLIOGRÁFICAS

BACON, J. e HARRIS, T. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

DEITELL, H. M. & DEITEL, P. J. & CHOHNES, D. R. *Sistemas Operacionais*. São Paulo, Ed. Prentice Hall, 2005.

MACHADO, F. e MAIA, L. *Arquitetura de Sistemas Operacionais*. 4^a Edição, LTC, 2007

OLIVEIRA, R. S. e CARISSIMI, A. S. e TOSCANI, S. S. *Sistemas Operacionais*. 3^a ed, volume 11, Ed. Bookman, 2008.

SHAW, A. C. *Sistemas e software de tempo real*. Porto Alegre, Ed. Bookman, 2003.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Fundamentos de Sistemas Operacionais*. 6^a Edição, Ed. LTC.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Sistemas Operacionais com Java*. 7^a Edição, Rio de Janeiro, Ed. Elsevier, 2008.

TANENBAUM, A. S. & WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*. 3^a Edição, Porto Alegre, Ed. Bookman, 2008.

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



O Sistema de Arquivo, sem sombra de dúvida, é a parte do Sistema Operacional mais visível para o usuário. Por se tratar da parte mais visível para usuários (alguns bastante experientes, outros, nem tanto), o sistema de Arquivo tem uma grande responsabilidade e desafio: apresentar uma interface fácil.

A maioria dos programas manipulam informações e necessitam grava-las de forma permanente (através de unidades chamadas **arquivos**). Outra questão que deve ser considerada é que, frequentemente, vários processos acessem as informações ou parte delas simultaneamente.

A parte do Sistema Operacional responsável por gerenciar os arquivos, o modo como eles são estruturados, nomeados, acessados, utilizados, protegidos e implementados é conhecida como o **Sistema de Arquivos** e será o assunto tratado nesta Unidade.

SUMÁRIO

UNIDADE V – SISTEMAS DE ARQUIVOS

1. INTRODUÇÃO AOS SISTEMAS DE ARQUIVOS

1.1. Arquivos

1.1.1. Nomeação de Arquivos

1.1.2. Estruturas de Arquivos

1.1.3. Tipos de Arquivos

1.1.4. Acesso aos Arquivos

1.1.5. Atributos de Arquivos

1.1.6. Operações com Arquivos

1.2. Implementação de Arquivos

1.2.1. Alocação Contígua

1.2.2. Alocação por Lista Encadeada

1.2.3. Alocação por Lista Encadeada Usando Índice

1.3. Diretórios

1.3.1. Organização de Sistemas de Diretórios

1.3.2. Nomes de Caminhos

1.3.3. Operações com Diretórios

1.4. Implementação de Diretórios

WEBLIOGRAFIA

REFERÊNCIAS BIBLIOGRÁFICAS

UNIDADE V

SISTEMAS DE ARQUIVOS

1. INTRODUÇÃO AOS SISTEMAS DE ARQUIVOS

Segundo Tanenbaum, existem três requisitos essenciais para o armazenamento de informação por longo prazo:

- Deve ser possível armazenar uma grande quantidade de informação;
- A informação deve sobreviver ao término do processo que a usa;
- Múltiplos processos têm de ser capazes de acessar a informação concorrentemente.

A solução encontrada usualmente para o armazenamento de informações é utilizar mídias externas e distintas da memória principal (volátil) em unidades chamadas **arquivos**, de modo que essas informações devem ser armazenadas de forma persistente, ou seja, não pode ser afetada pela criação e término de um processo. Assim, um arquivo só irá ser destruído quando seu proprietário removê-lo explicitamente.

Os **arquivos** são recipientes de dados, identificados por um nome e por uma série de atributos, mantidos e gerenciados pelo Sistema Operacional.

Já os **diretórios** são conjuntos de referências a arquivos. Os diretórios permitem agrupar arquivos, facilitando manuseio e localização.

Do ponto de vista do usuário, o aspecto mais importante do Sistema de Arquivos é como aparece para ele, ou seja, como o Sistema de Arquivos é implementado e quais estruturas são utilizadas não é a questão primordial. O usuário está mais interessado no que o arquivo é constituído, nomeados e protegidos.

1.1. Arquivos

Neste tópico veremos os arquivos do ponto de vista do usuário, abordando suas principais propriedades. É importante lembrar que este capítulo irá apresentar as características gerais de Sistemas de Arquivos, de uma forma geral. Porém, cada Sistema de Arquivo específico contém características próprias. Caso o leitor tenha interesse em obter mais informações de um determinado Sistema de Arquivo específico, a *webliografia* desta unidade traz alguns *links* que direcionam à sites com informações dos Sistemas de Arquivos mais conhecidos: FAT32, NTFS e EXT.

1.1.1. Nomeação de Arquivos

Segundo Tanenbaum, o arquivo é um mecanismo de abstração, no qual oferece meios de armazenar informações no disco e lê-las posteriormente. A característica mais importante de qualquer mecanismo de abstração é o modo como os objetos são gerenciados e nomeados, isto é, a nomeação de arquivos.

Um processo ao criar um arquivo, atribui um nome a ele e quando o processo termina, este arquivo continua existindo, possibilitando que outros processos possam ter acesso a ele simplesmente buscando por seu nome.

As regras de nomeação de arquivos variam de sistema para sistema, porém, como característica comum, todos os Sistemas Operacionais atuais permitem cadeias de caracteres de um até oito letras como nomes válidos de arquivos. Alguns Sistemas de Arquivos permite a utilização de caracteres especiais, outros não; alguns Sistemas de Arquivos permitem nomes com tamanhos até 255 caracteres; alguns Sistemas de Arquivos fazem distinção entre letras maiúsculas e minúsculas (*case sensitive*), como exemplo o Sistema de Arquivo do Unix, enquanto que outros não diferenciam, como exemplo o Sistema de Arquivo do MS-DOS e do Windows.

A maioria dos Sistemas Operacionais (na verdade, o Sistema de Arquivo do SO) suporta nomes de arquivos de duas partes, separados por um ponto. A parte que segue o ponto é chamada de

extensão do arquivo e indica o tipo do arquivo. Essa extensão, em alguns Sistemas Operacionais, é utilizada para atribuir qual programa deve abrir aquele arquivo em específico.

1.1.2. Estruturas de Arquivos

Segundo Tanenbaum, os arquivos podem ser estruturados, basicamente, através de três maneiras, como visualizados na Figura 43.

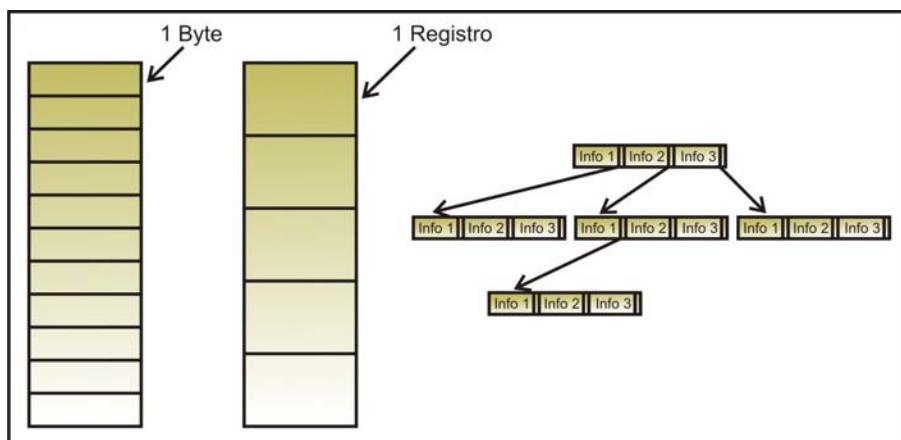


Figura 43: Estruturas de Arquivos.

- Seqüência de bytes não-estruturada: nessa estrutura o SO não sabe o que o arquivo contém e tudo que ele enxerga é uma seqüência de bytes. Tal estratégia apresenta uma máxima flexibilidade, uma vez que os programas dos usuários podem pôr qualquer coisa que queiram em seus arquivos e chamá-los do nome que lhes convier.
- Seqüência de registro de comprimento fixo: Um arquivo é uma seqüência de registros de tamanho fixo, cada um com alguma estrutura interna. A idéia central é que a operação de leitura retorna um registro e a operação de escrita sobrepõe ou anexa um registro.
- Árvore de Registros: um arquivo é constituído de uma árvore de registros (não necessariamente do mesmo

tamanho), cada um contendo um campo-chave em uma posição fixa no registro, na qual a árvore é ordenada pelo campo-chave para que se busque mais rapidamente por uma chave específica. Além disso, novos registros podem ser adicionados ao arquivo, decidindo o sistema operacional onde colocá-los. Este tipo de arquivo é amplamente aplicado em computadores de grande porte usados ainda para alguns processamentos de dados comerciais.

1.1.3. Tipos de Arquivos

Vários tipos de arquivos são suportados pelos Sistemas Operacionais. Podemos enumerar os principais tipos:

- **Arquivos comuns:** arquivos que contêm informações do usuário.
- **Diretórios:** arquivos do sistema que mantêm a estrutura do sistema de arquivos.
- **Arquivos especiais de caracteres:** arquivos relacionados à entrada e saída e usados para modelar dispositivos de E/S.
- **Arquivos especiais de blocos:** arquivos usados para modelar discos.

Os arquivos comuns (informações do usuário), em geral, são arquivos ASCII ou arquivos binários. Os arquivos ASCII são constituídos de linhas de texto, possuindo como grande vantagem o fato de que eles podem ser mostrados e impressos como são e poder ser editados com qualquer editor de textos. Além disso, facilita a conexão entre a saída de um programa e a entrada de um outro.

Já os arquivos binários, em geral, possuem alguma estrutura interna, conhecida pelos programas que os usam. Todo Sistema Operacional deve reconhecer pelo menos um tipo de arquivo: seu próprio arquivo executável.

1.1.4. Acesso aos Arquivos

O acesso a arquivos pode ser realizado, basicamente, através de duas maneiras.

- **Acesso seqüencial:** nesse tipo de acesso os arquivos são lidos em ordem sequencialmente, partindo do início, mas nunca saltando e lendo fora de ordem. Esse tipo de acesso é muito utilizado, como exemplo, leitura de mídias do tipo fita magnética, além de leitura sequencial de programas fontes, realizada por vários compiladores.
- **Acesso Aleatório:** em alguns tipos de mídias é possível ler bytes ou registros de um arquivo fora da ordem. Esses arquivos são chamados de **arquivos de acesso aleatório**.

Segundo Tanenbaum, em alguns Sistemas Operacionais antigos, os arquivos eram classificados por acesso sequencial ou por acesso aleatório no momento em que estavam sendo criados. Entretanto, os Sistemas Operacionais modernos não fazem distinção, ou seja, todos os seus arquivos são, automaticamente, de acesso aleatório.

1.1.5. Atributos de Arquivos

A maioria dos Sistemas Operacionais associa várias informações extras a um determinado arquivo. Essas informações são, comumente, chamadas de atributos de um arquivo.

Os tipos de atributos variam de sistema para sistema e nenhum sistema dispõe de todos os tipos de atributos possíveis. Podemos enumerar uma série de atributos, como: pessoas que podem acessar, senhas, criador, proprietário, comprimento do arquivo, tempo de criação, tempo do último acesso, tempo da última atualização, tamanho máximo, dentre outros.

1.1.6. Operações com Arquivos

A finalidade dos arquivos é armazenar informação e permitir que ela seja recuperada posteriormente. Assim, os Sistemas de Arquivos oferecem diferentes operações para armazenar e recuperar

informações. Dentre as várias operações possíveis sobre arquivos, podemos destacar:

- Criar: operação utilizada para criar o arquivo.
- Apagar: operação utilizada para excluir um determinado arquivo, liberando espaço de memória.
- Abrir: operação utilizada para abrir um arquivo e permitir escrita ou leitura de informações.
- Fechar: operação utilizada para fechar um arquivo e manter sua consistência.
- Ler: operação disponibilizada para ler informações do arquivo, para poderem ser utilizadas.
- Escrever: operação para incluir alguma informação no arquivo.

1.2. Implementação de Arquivos

Vários métodos são utilizados para implementar arquivos e neste tópicos examinaremos alguns deles.

1.2.1. Alocação Contígua

Trata-se do exemplo mais simples de implementar, no qual é utilizado um bloco de dados contíguo (sequencial) no disco para armazenar um arquivo. Por exemplo, considerando um disco com blocos de 1k, um arquivo com 20k ocuparia 20 blocos consecutivos.

Este tipo de alocação traz como principal vantagens a simplicidade de implementação e o fato de ter um bom desempenho no acesso a um arquivo, pois um arquivo inteiro pode ser lido com apenas uma operação.

Porém, a alocação contígua possui grandes desvantagens. A primeira está no fato de que este tipo de alocação só é praticável caso seja conhecido o tamanho do arquivo no momento de sua criação. Outra desvantagem é que este tipo de alocação gera uma fragmentação no disco.

A alocação contígua é representada através da Figura 44.

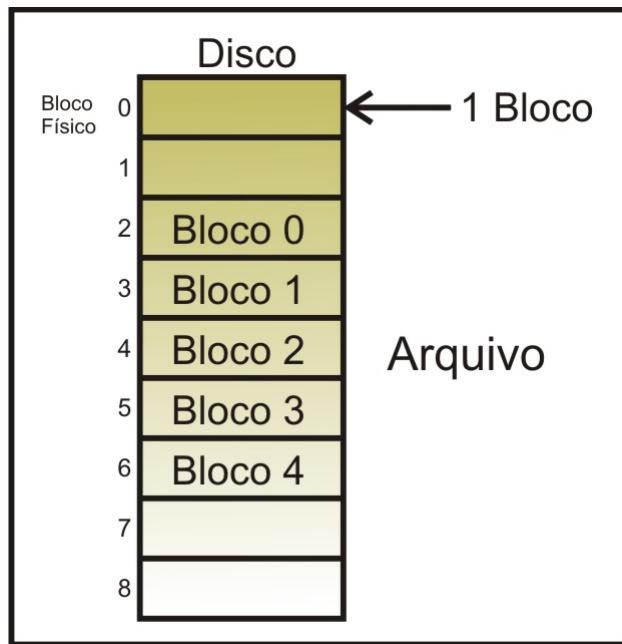


Figura 44: Implementação de Arquivos por Alocação Contígua.

1.2.2. Alocação por Lista Encadeada

Outro método para armazenar um arquivo é utilizando listas encadeadas de blocos de disco. Cada bloco conterá, dessa forma, o seu dado armazenado e um ponteiro para o bloco seguinte.

A principal vantagem de se utilizar a alocação por lista encadeada está no fato de evita o grande desperdício em disco, comum na alocação contígua. Além disso, para acessar todo o arquivo é suficiente armazenar apenas o endereço de disco do primeiro bloco.

Porém, a principal desvantagem da utilização desse método é que o acesso aleatório é extremamente lento. Além disso, considerando que cada bloco terá que guardar o endereço do bloco seguinte, existe um gasto de memória adicional. A alocação por lista encadeada é representada através da Figura 45.

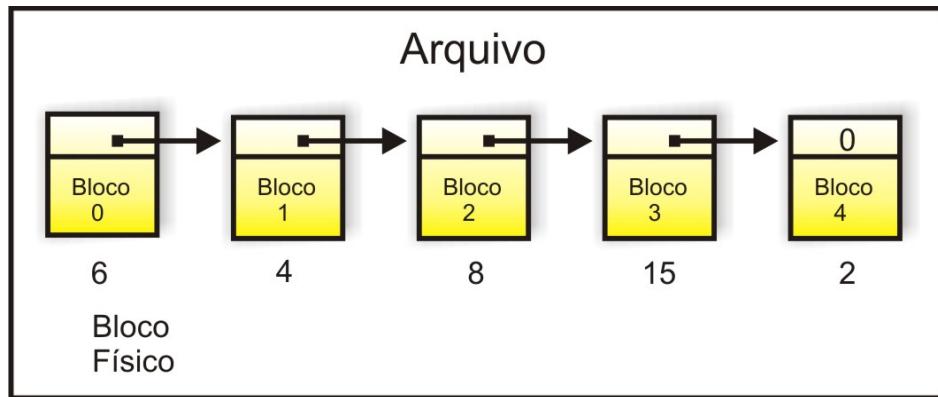


Figura 45: Implementação de Arquivos por Alocação por Lista Encadeada.

1.2.3. Alocação por Lista Encadeada Usando Índice

Uma maneira de tentar driblar as duas desvantagens de se utilizar alocação por lista encadeada seria utilizar a palavra ponteiro de cada bloco de disco de um arquivo e colocar em uma tabela ou em um índice na memória.

Dessa forma, o bloco inteiro no disco estará disponível para armazenamento de dados (não é necessário armazenar mais o ponteiro para o próximo bloco). Além disso, o acesso aleatório é muito mais fácil, pois a cadeia de blocos está toda em memória, não sendo necessário qualquer referência ao disco. A Figura 46 ilustra a tabela utilizada neste método de alocação para armazenar a sequência dos blocos do arquivo representado pela lista encadeada da Figura 45.

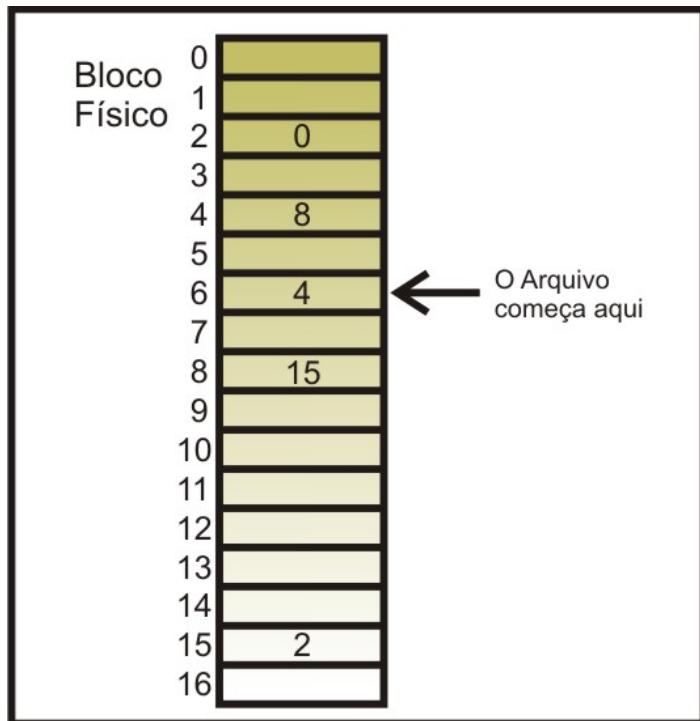


Figura 46: Implementação de Arquivos por Alocação por Lista Encadeada, utilizando índice.

1.3. Diretórios

Os sistemas de arquivos têm, em geral, **diretórios** ou **pastas** para controlar os arquivos, que, em muitos sistemas, também são considerados arquivos.

1.3.1. Organização de Sistemas de Diretórios

A maneira mais simples de se projetar um Sistema de Arquivos é ter um diretório contendo todos os arquivos, comumente chamado de diretório-raiz.

Esse sistema era muito comum, em parte, nos primeiros computadores pessoais, pois havia somente um usuário. Porém, em um sistema com vários usuários o problema de haver somente um diretório é que diferentes usuários podem usar acidentalmente os mesmos nomes para seus arquivos. Em consequência disso, esse esquema não é mais empregado em sistemas multusuário, mas,

comumente empregado em sistemas embarcados (*embedded Systems*). Esse esquema é demonstrado na Figura 47.

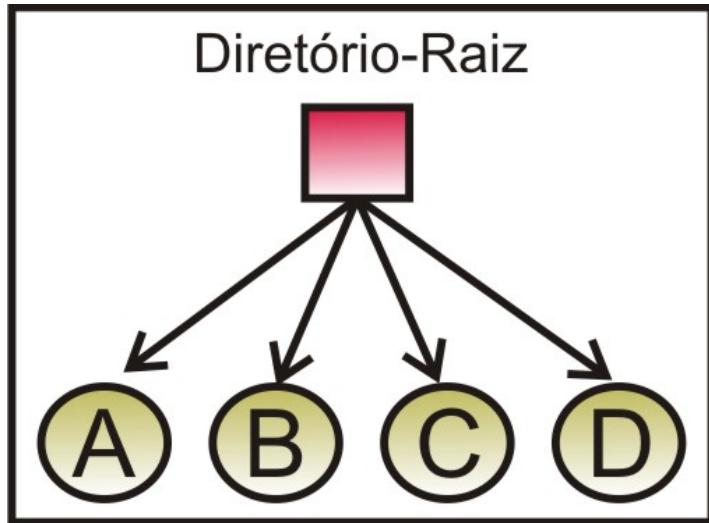


Figura 47: Projeto de Sistemas de Arquivos com um nível.

Com a finalidade de evitar conflitos causados por diferentes usuários escolhendo o mesmo nome para seus arquivos, a solução é oferecer um diretório privado para cada usuário. Dessa forma, os nomes escolhidos por um usuário não interferiria nos nomes escolhidos por outro usuário, podendo ocorrer de arquivos com o mesmo nome em dois ou mais diretórios sem causar nenhum problema. Este esquema está ilustrado na Figura 48.

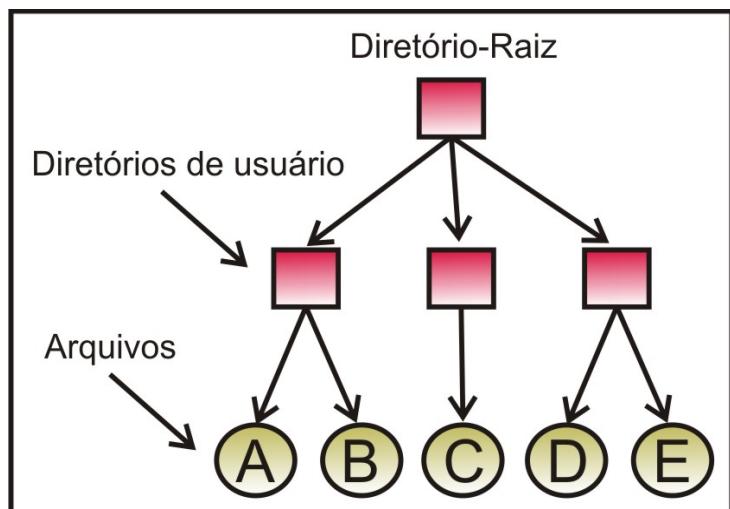


Figura 48: Projeto de Sistemas de Arquivos com dois níveis.

Os conflitos de nomes entre os usuários são eliminados na hierarquia em dois níveis, mas tal hierarquia não é satisfatória para os usuários com um número muito grande de arquivos ou que desejam agrupar seus arquivos de maneira lógica. Dessa forma, podemos utilizar uma hierarquia geral, ou seja, uma árvore de diretórios, no qual cada usuário pode ter tantos diretórios quanto necessários para agrupar os seus arquivos de uma maneira natural. Assim, essa capacidade dos usuários criarem um número arbitrário de subdiretórios constitui uma poderosa ferramenta de estruturação para organizar seus trabalhos. Por esse motivo, quase todos os modernos sistemas de arquivos são organizados dessa forma. A Figura 49 ilustra este tipo de projeto hierárquico.

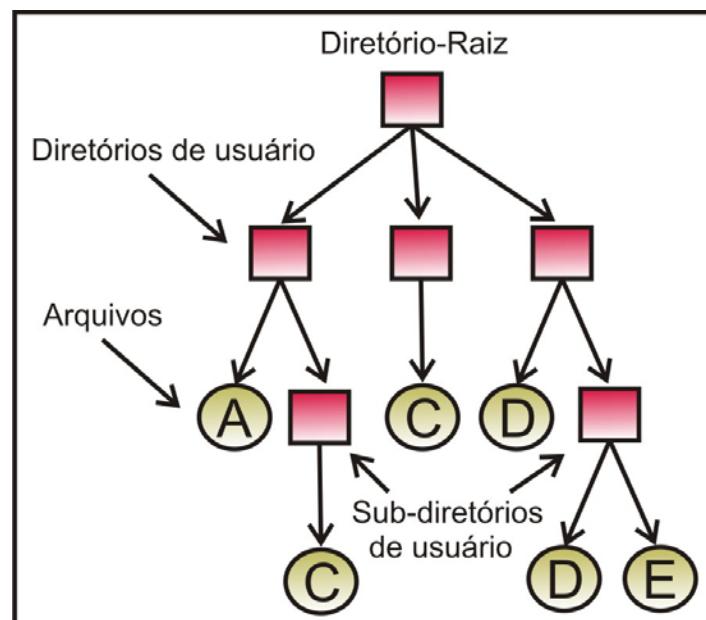


Figura 49: Projeto de Sistemas de Arquivos hierárquico.

1.3.2. Nomes de Caminhos

Quando organizamos um Sistema de arquivos baseado em uma árvore de diretórios, torna-se necessário uma forma de especificar o nome dos arquivos, e para isso são usados, comumente, dois métodos: **nome de caminho absoluto** e **nome de caminho relativo**.

Um nome de caminho absoluto é formado pelo caminho entre o diretório-raiz e o arquivo específico. Os nomes de caminhos absolutos sempre iniciam no diretório-raiz e são únicos. Em Sistemas de Arquivos do *Windows* os componentes do caminho são separados por \. Já em Sistemas de Arquivos do *Unix* são separados por /.

Já o nome de caminho relativo é utilizado em conjunto com o conceito de diretório de trabalho (também chamado diretório atual). Assim, um usuário pode designar um diretório de trabalho específico e quaisquer nomes de caminhos que não comecem a partir do diretório-raiz são interpretados em relação ao diretório do trabalho. Como exemplo, imagine que o diretório de trabalho de um usuário *UNIX* seja /home/usuário, então o arquivo, cujo nome de caminho absoluto seja /home/usuário/arquivo.txt pode, simplesmente, ser interpretado como arquivo.txt.

Cada processo possui seu próprio diretório de trabalho. Dessa forma, quando um processo altera seu diretório de trabalho e depois sai, nenhum outro processo é afetado e nenhum vestígio da mudança é deixado no sistema de arquivos. Por outro lado, procedimentos de bibliotecas raramente alteram o diretório de trabalho e, quando precisam fazê-lo, eles sempre voltam para onde estavam, se não o resto do programa poderá não funcionar.

A maioria dos Sistemas Operacionais que suportam um sistema de diretório hierárquico tem duas entradas especiais em cada diretório: ponto (.) e o ponto-ponto (..). O ponto refere-se ao diretório atual, enquanto que o ponto-ponto refere-se ao diretório pai (diretório anterior). Como exemplo, se um usuário tiver trabalhando no diretório /home/usuário, caso utilize o ponto-ponto, ele estará subindo na árvore de diretórios, ou seja, referenciando o diretório /home. Se ele utilizar o ponto, estará referenciando o próprio diretório (/home/usuário).

1.3.3. Operações com Diretórios

Como os arquivos, existem diversas chamadas ao sistema para gerenciar diretórios, que, também, variam de sistema para sistema. As operações mais importantes são:

- Criar: utilizado para criar um determinado diretório. Inicialmente é criado um diretório vazio, contendo apenas o ponto e o ponto-ponto.
- Apagar: operação utilizada para apagar um determinado diretório. Normalmente, só é possível excluir um diretório vazio.
- Abrir diretório: a partir desta operação é possível abrir um diretório. Antes de ler um diretório é preciso, inicialmente, abri-lo.
- Fechar diretório: operação utilizada para fechar um diretório.
- Ler: esta chamada é utilizada para ler o conteúdo de um diretório.
- Renomear: fornece a possibilidade de alterar o nome do diretório.

1.4. Implementação de Diretórios

O Sistema Operacional utiliza o nome de caminho fornecido pelo usuário para poder localizar a entrada de diretório. A entrada de diretório contém informações necessárias para a localização dos blocos de disco. Essas informações podem ser o endereço do arquivo inteiro ou o número do primeiro bloco, por exemplo. Assim, a principal função do sistema de diretório é mapear o nome ASCII do arquivo para as informações necessárias para localizar os dados em disco (Tanenbaum).

WEBLIOGRAFIA

Universidade Aberta do Piauí – UAPI

www.ufpi.br/uapi

Universidade Aberta do Brasil – UAB

www.uab.gov.br

Secretaria de Educação à Distância do MEC - SEED

www.seed.mec.gov.br

Associação Brasileira de Educação à Distância – ABED

www.abed.org.br

Curso de Sistemas Operacionais – DCA/UFRN

Prof. Dr. Luiz Affonso Henderson Guedes de Oliveira

<http://www.dca.ufrn.br/~affonso/DCA0108/curso.html>

Home Page do Prof. Dr. Rômulo Silva de Oliveira

<http://www.das.ufsc.br/~romulo/>

Home Page do Autor Andrew S. Tanenbaum

<http://www.cs.vu.nl/~ast/>

Simulador de Ensino para Sistemas Operacionais

<http://www.training.com.br/sosim/>

NTFS – *New Technology File System*

<http://www.ntfs.com/>

FAT32 *File System Specification*

<http://www.microsoft.com/whdc/system/platform/firmware/fatgen.msp>

X

Ext2fs Home Page

<http://e2fsprogs.sourceforge.net/ext2.html>

REFERÊNCIAS BIBLIOGRÁFICAS

BACON, J. e HARRIS, T. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

DEITELL, H. M. & DEITEL, P. J. & CHOHNES, D. R. *Sistemas Operacionais*. São Paulo, Ed. Prentice Hall, 2005.

MACHADO, F. e MAIA, L. *Arquitetura de Sistemas Operacionais*. 4^a Edição, LTC, 2007

OLIVEIRA, R. S. e CARISSIMI, A. S. e TOSCANI, S. S. *Sistemas Operacionais*. 3^a ed, volume 11, Ed. Bookman, 2008.

SHAW, A. C. *Sistemas e software de tempo real*. Porto Alegre, Ed. Bookman, 2003.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Fundamentos de Sistemas Operacionais*. 6^a Edição, Ed. LTC.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. *Sistemas Operacionais com Java*. 7^a Edição, Rio de Janeiro, Ed. Elsevier, 2008.

TANENBAUM, A. S. & WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*. 3^a Edição, Porto Alegre, Ed. Bookman, 2008.

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

SOBRE O AUTOR

Erico Meneses Leão



Possui Graduação em Bacharelado em Ciência da Computação pela Universidade Federal do Piauí, Brasil, em 2005; é Mestre em Ciências, pelo Programa de Pós-Graduação em Engenharia Elétrica, com ênfase em Engenharia de Computação, da Universidade Federal do Rio Grande do Norte, Brasil, em 2007. Foi professor do curso de Ciência da Computação, do Centro de Ensino Unificado de Teresina – CEUT, e do curso de Sistema de Informação, da Faculdade de Atividades Empresariais de Teresina - FAETE. Desde 2008, é professor do quadro efetivo, e Assistente do Departamento de Informática e Estatística – DIE, da Universidade Federal do Piauí - UFPI. Seus principais interesses de pesquisa e atuação incluem arquitetura de sistemas distribuídos de tempo real, sistemas operacionais, redes de computadores e sistemas de comunicação de tempo real para redes industriais.