APOSTILA

de

SISTEMAS

OPERACIONAIS

Índice

1 INTRODUÇÃO AO SISTEMA OPERACIONAL

Podemos dizer sem receio que um computador sem *software* não passa de peso para papel. Talvez a prova mais evidente nos dias atuais é o sucesso nos sistemas operacionais da Microsoft Corp. O grande motivo deste sucesso, apesar de muitas pessoas de renome afirmarem que os sistemas operacionais da Microsoft não são tecnicamente bons, deve-se à enorme quantidade de *software* disponível para estes sistemas operacionais.

Mas afinal, se é importante para as pessoas a existência de bons *softwares* que ajudem nos seus trabalhos, como pode o sistema operacional influenciar na qualidade e na disponibilidade de tais *softwares*?

Para responder esta pergunta, precisamos definir o que é um sistema operacional. Ele nada mais é do que um programa de computador, que após o processo de inicialização (*boot*) da máquina, é o primeiro a ser carregado, e que possui duas tarefas básicas:

- gerenciar os recursos de *hardware* de forma que sejam utilizados da melhor forma possível, ou seja, "tirar" o máximo proveito da máquina fazendo com que seus componentes estejam a maior parte do tempo ocupados com tarefas existentes; e
- prover funções básicas para que programas de computador possam ser escritos com maior facilidade, de modo que os programas não precisem conhecer detalhes da máquina para poderem funcionar.

É justamente neste segundo item que os sistemas operacionais podem ser bem sucedidos ou não, em despertar interesse para que a indústria de *software* e os programadores independentes construam programas para determinados sistemas operacionais. Isto justifica parte do sucesso do Microsoft Windows, pois, ao mesmo tempo que ele provê uma interface bastante amigável com o usuário, para o programador, não é tão difícil criar um programa com janelas, botões, listas, etc, como seria num sistema operacional como o MS-DOS. Além disso, os sistemas operacionais da Microsoft rodam no *hardware* mais "popular" hoje em dia: os computadores baseados em IBM PC.

Computadores modernos possuem um ou mais processadores, memória principal, dispositivos de entrada e saída como discos, fitas, teclado, mouse, monitor, interface de rede, entre outros. Escrever programas que utilizem um computador com esta complexidade de forma eficiente é muito difícil e trabalhoso. É exatamente neste ponto que entram as funções do sistema operacional: abstrair as particularidades do *hardware* dos programas, fornecendo a eles facilidades para sua operação, tais como: rotinas de acesso a dispositivos diversos; funções de armazenamento de dados como criação de arquivos, leitura e escrita de dados; e rotinas de acesso aos dispositivos de interação com a máquina, como teclado, *mouse*, monitor, etc.

Dada a existência de *softwares* como o sistema operacional, os programas normalmente são classificados como *software* básico (que inclui o sistema operacional), e *softwares* de aplicação, que são voltados a resolver problemas dos usuários.
Podemos visualizar através de um diagrama a integração entre *hardware*, *software*

básico, e *softwares* aplicativos, como mostra a figura 1.1.

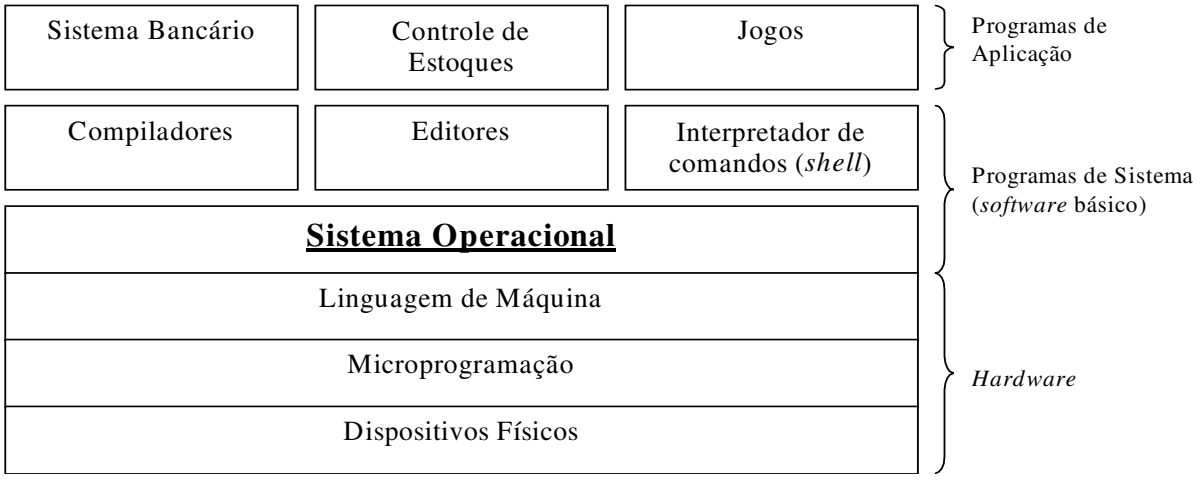


Figura 1.1 – Integração entre *hardware*, *software* básico e *software* aplicativo

Olhando para o diagrama, veremos que o que chamamos de "*hardware*" é na verdade composto de três camadas. Nem todas as máquinas seguem este esquema, algumas podem ter uma camada a menos, ou mesmo camadas adicionais, mas basicamente, os computadores seguem o esquema ilustrado na figura 1.1.

No nível mais inferior, temos os dispositivos eletrônicos em si, como o processador, os *chips* de memória, controladores de disco, teclado, e outros dispositivos, barramentos, e qualquer dispositivo adicional necessário para o funcionamento do computador. Um nível acima, temos a camada de "microprogramação", que de forma geral, são pequenos passos (chamados de microoperações) que formam uma instrução de processador completa, como ADD, MOV, JMP, etc.

O conjunto de instruções do computador é chamado de linguagem de máquina, e apesar de ser uma espécie de linguagem, podemos dizer que faz parte do *hardware* porque os fabricantes a incluem na especificação do processador, para que os programas possam ser escritos. Afinal, de nada adianta uma máquina maravilhosa, se não existir documentação alguma de como ela funciona. Assim, as instruções que a máquina entende são consideradas parte integrante do *hardware*.

As instruções também incluem, geralmente, operações que permitem ao processador comunicar-se com o mundo externo, como controladores de disco, memória, teclado, etc. Como a complexidade para acesso a dispositivos é muito grande, é tarefa do Sistema Operacional "esconder" estes detalhes dos programas. Assim, o sistema operacional pode, por exemplo, oferecer aos programas uma função do tipo "LEIA UM BLOCO DE UM ARQUIVO", e os detalhes de como fazer isso ficam a cargo do sistema operacional.

Acima do sistema operacional estão os demais programas utilizados pelo usuário final, mas alguns deles ainda são considerados *software* básico, como o sistema operacional. Entre eles podemos citar o *shell*, que consiste do interpretador de comandos do usuário, ou seja, a interface com o usuário. Nos sistemas operacionais

mais recentes, freqüentemente o *shell* é uma interface gráfica (ou em inglês GUI – *Graphics User Interface*). Raramente, numa interface gráfica bem elaborada, o usuário precisa digitar comandos para o computador. A maneira mais comuns de executar programas, copiar e mover arquivos, entre outras atividades mais comuns, é através do uso do *mouse*. Nos tempos do MS-DOS, o teclado era o dispositivo de entrada dominante, por onde o usuário entrava todos os comandos para realizar suas tarefas do dia a dia.

O que é muito importante observar quanto ao *software* básico é que, apesar de que editores (ex: bloco de notas do Windows), compiladores (ex: compilador C no Unix), e interpretadores de comando (ex: command.com ou explorer.exe no Windows) normalmente serem instalados junto como sistema operacional em um computador, eles não são o sistema operacional. Eles apenas utilizam o sistema operacional. Portanto, o *shell* que normalmente usamos em um sistema operacional nada mais é do que um programa que utiliza serviços do sistema operacional, mas com a finalidade de permitir que os usuários realizem suas tarefas mais freqüentes: executar programas e trabalhar com arquivos.

A grande diferença entre o sistema operacional, e os programas que rodam sobre ele, sejam *software* básico ou *software* aplicativo, é que o sistema operacional roda em modo kernel (ou supervisor), enquanto os demais programas rodam em modo usuário. Estes dois modos de operação dos processadores dos computadores diferem no fato de que em modo supervisor, um programa tem acesso a todo o *hardware*, enquanto que os programas que rodam em modo usuário, tem acesso somente a determinadas regiões de memória, não podem acessar dispositivos diretamente, e precisam pedir para o sistema operacional quando necessitam de alguma tarefa especial. Isto garante que os programas dos usuários, não acabem por invadir áreas de memória do sistema operacional, e acabem por "travar" o sistema. Isto também possibilita que programas de diferentes usuários estejam rodando na mesma máquina, de forma que um usuário não consiga interferir nos programas de outro.

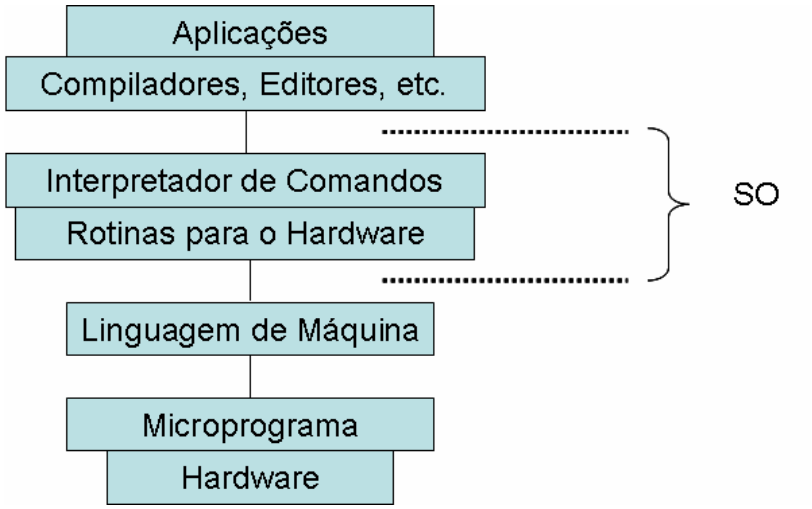
1.1 O que é um Sistema Operacional?

• Uma camada de software muito próxima ao hardware, e que contem instruções privilegiadas e permite ao menos dois modos de operação (modo kernel ou protegido, e modo usuário).

• Uma das máquinas virtuais de um sistema multi-níveis que provê abstrações adequadas aos programadores e usuários.

• Um "isolador" entre os programadores e o hardware que contem as rotinas de controle do hardware.

• Uma forma de permitir a implementação de linguagens de alto-nível e contem as rotinas em comum para E/S e oferece segurança.



Do ponto de vista das aplicações o Sistema Operacional se apresenta como uma interface conveniente, e do ponto de vista do hardware, como um gerente de recursos. Isso atende às duas finalidades pelas quais historicamente os SO foram criados: tornar os complexos computadores em máquinas convenientes para o usuário, e em máquinas eficientes para a resolução de problemas. Entre as diversas definições para os SO's uma que parece sensata é:

"Sistemas Operacionais são programas de controle dos recursos do computador, gerenciando eventuais conflitos, e alocando esses recursos da maneira mais eficiente possível. Constituem assim uma forma razoável de tornar os complexos componentes do hardware em algo utilizável na execução de tarefas para o usuário."

1.2 Tipos de Sistemas Operacionais

1.2.1 Sistemas Operacionais em Lote (ou Batch)

São os mais antigos, pois apareceram por volta de 1956, na época dos cartões perfurados. Pode-se dizer que seu surgimento se deveu ao uso de alguns cartões de controle.

Sua principal característica é a falta de qualquer, tipo de interação com o usuário. As tarefas, ou "jobs", são agrupadas conforme seu tipo, geralmente determinado pela linguagem-fonte, e executadas sequencialmente uma-a-uma.

Sua finalidade é minimizar o tempo ocioso de CPU e de periféricos, devido ao seu elevado custo, à custa de grandes tempos de resposta. Os tempos do processamento (médios) são razoavelmente pequenos, já que o n° de jobs/unidade de tempo

consegue ser bastante elevado (por exemplo: 156 jobs/dia ou 6,5 jobs/hora, ou, em média = 9,23 minutos de processamento para cada job).

É importante observar que nos sistemas do tipo batch os conceitos de I/O bound e tarefas CPU bound (surgidos também nessa época) podem ser usados para agrupar os jobs. O ideal é uma combinação de tarefas de características opostas, para permitir ocupação e velocidade máximas tanto da CPU quanto dos periféricos.

Observação:

Nestes sistemas operacionais o objetivo é maximizar o throughput (jubas/unidade de tempo) à custa de maiores tempos de resposta.

1.2.2 sistemas operacionais interativos (ou time-sharing)

Ao contrário dos sistemas batch, cujos longos tempos de resposta são um desincentivo ao desenvolvimento e à criatividade, nos sistemas T.S. a interação com ò usuário é freqüente, à custa de tempos de processamento mais longos. Os pequenos tempos de resposta necessários a essa interação constante exigem que periodicamente o sistema operacional seja executado, expulsando o processo corrente da CPU, o que é chamado de time-slicing, ou concorrência.

Diversos algoritmos, que serão vistos posteriormente, visam melhorar a eficiência do time-slicing,.dedicando, por exemplo, o tempo de processamento de um usuário ao processamento de tarefas de outro. Isto permite grande realocação de recursos, inclusive memória e CPU, minimizando a ociosidade.

1.2.3 Sistemas Operacionais de Tempo Real (ou Real-Time)

Sua aplicação típica é em controle de processos industriais. Necessitam, no mínimo, de relógio de tempo real e capacidade de conversão D/A-A/D, e suas exigências de tempo de resposta são rígidas.

Os mecanismos de interrupção desses sistemas devem ser muito eficientes, e preemptivos (ou seja: uma rotina de tratamento de interrupção pode ser bloqueada para dar à vez a outra mais prioritária, o que equivale a dizer que as interrupções podem ser aninhadas).

O grau de multiprogramação é elevado, o que significa que a concorrência à CPU (vários processos tentando "rodar") é grande.

A tolerância a falhas também deve ser bem projetada, ou seja a capacidade de de-

tecção, confinamento, recuperação, e operação parcial, bem como a capacidade de comunicação assíncrona (ou seja, não necessitar de uma resposta a uma solicitação qualquer para poder continuar a execução, rnesmo que degradada).

A detecção de erros se refere à capacidade de identificar, de forma rápida e não ambígua, a fonte do erro, o confinamento se refere à capacidade de não permitir que a falha se propague e afete outros componentes, a operação parcial se refere a poder continuar operando, mesmo que de forma mais limitada, sem que isso comprometa o sistema como um todo, e a recuperação se refere à assistência prestada aos encarregados da manutenção do sistema e aos usuários.

1.2.4 Programas Monitores

São programas muito simples, usados em geral em computadores pequenos ou experimentais, muitas vezes dedicados. Contem o interpretador de comandos e rotinas simples para controle do hardware.

Os comandos do interpretador nunca diferem muito de:

- Listar, alterar, mover blocos de memória.
- Executar programas do usuário, sequencialmente ou passo-a-passo.
- Incluir/excluir breakpoints, para uso dos depuradores.
- Realizar E/S e controlar periféricos simples.

Em geral todos esses serviços são prestados com rotinas de nível muito baixo, tipicamente linguagem Assembly. Sua execução segue um ciclo do tipo espera, interpreta, executa, responde, espera, interpreta, executa, responde...

1.3 Serviços de um Sistema Operacional

a) Objetivos de um Sistema Operacional

- Compartilhar o hardware com eficiência e confiabilidade. A grande dificuldade é a imprevisibilidade de demanda. Usam-se, na prática, determinações estatísticas da carga de trabalho típica.
- .Proteger um usuário das ações de outros, sejam essas ações intencionalmente maldosas ou não. Uma saída é fornecer a cada usuário uma máquina

virtual independente, mas isso dificulta a comunicação quando ela se fizer necessária.

- Reduzir os efeitos de falhas de hardware ou software.

- Apresentar tempos de resposta previsíveis.

É evidente que alguns desses objetivos são bastante difíceis de se alcançar, principalmente em conjunto. Na verdade o objeto principal de nosso estudo é compreender exatamente os problemas que surgem quando tentamos conciliá-los.

b) Objetivos do Ponto de Vista do Usuário

- Executar programas: significa poder carregá-los para a memória, dar-lhes, ó controle e poder recebê-lo de volta, e atender suas chamadas em geral.

- Realizar E/S: o sistema operacional deve conter os "drivers" que conhecem os periféricos, para que o usuário não precise se preocupar com detalhes técnicos (tipo a posição de memória).

- Criar e manter um sistema de arquivo: converter detalhes de hardware como trilhas, setores, “clusters”.

- Fornecer apoio pré e pós-falhas: permitir o uso do computador com segurança, procurando evitar ou minimizar o efeito de falhas de hardware e software, e enviando mensagens de erro e recomendações úteis.

c) Objetivos do Ponto de Vista do Hardware

- Alocar recursos com eficiência, minimizando a ociosidade de quaisquer componentes.

- Manter a contabilidade dos recursos, tanto para fins estritamente comerciais como para fins de auditoria ou de uso do próprio sistema.

- Prover mecanismos de proteção em software (por exemplo: rotinas para tratamento de "traps").

1.4 Funções de um Sistema Operacional

Ao contrário dos objetivos do SO, que se referem ao que ele deve ser propor a reali-

zar, as funções dizem respeito à forma de realizar esses objetivos.

a) Do Ponto de Vista do Usuário

Chamadas ao Sistema: são rotinas de nível muito baixo, em geral com formato de linguagem Assembly. São a forma mais simples de interface entre usuário (ou seus programas) e o hardware. Podem ser dos tipos:

- Controle de jobs - término normal ou anormal de execução, criação de outro processo, definição ou alteração de prioridades, comunicação entre processos, informação ou alteração de perfil de execução (tempo, localidade em memória, quantidade de E/S, interrupções e traps).

- Manipulação de arquivos - criar, mover ponteiros de leitura/escrita, abrir, fechar, ler, escrever seqüencial/randômico, remover, sempre a partir de nome e atributos.

- Gerência de dispositivos - requisitar, liberar, ler/escrever, redirecionar (lembrando que arquivos são dispositivos virtuais), alterar atributos.

- Informações de/para o sistema – data, hora, número de usuários, versão do S.O., capacidade livre de memória e disco, atributos.

A implementação dessas chamadas pode ser feita na forma de instruções SVC tipo onde define o serviço solicitado (e pode ser operando imediato da instrução, ou um registrador da CPU como operando ou como ponteiro).

Para copiar um arquivo para outro, por exemplo, são necessárias duas chamadas para ler os nomes dos arquivos no teclado, duas chamadas para abri-los (supondo que não haja nenhum erro na abertura), duas chamadas para o loop de leitura/escrita e duas chamadas para fechá-los.. No caso da ocorrência de erros (do tipo: arquivo fonte inexistente, arquivo destino já existente, arquivos protegidos, erro geral de leitura ou escrita, etc.) teremos mais uma chamada para cada condição e mais uma para término anormal.

Programas do Sistema – são de nível mais alto que as chamadas. Os detalhes de abrir, fechar, etc., típicos em chamadas ficam escondidos do usuário. Para copiar, por exemplo, o próprio programa se encarregaria de fazer todo o tratamento dos parâmetros necessários, e apenas solicitaria ao usuário os nomes dos arquivos, retornando-lhe ao final algum tipo de mensagem. Esses programas são dos tipos:

- Manipulação de arquivos e diretórios.

- Informações e status: data, hora, área livre em memória ou disco, número de usuários.

- Suporte a linguagens de alto nível: carga, depuração e execução de programas.

- Pacotes de aplicações: gerenciadores de bancos de dados, editores de texto, gerenciadores de impressão, etc.

Interpretador de comandos - procura tornar a interface com o usuário o mais conversacional possível (o que não significa que seja amigável), procurando ser um programa de sistema que se encarrega de chamar outros programas, sejam eles do sistema ou do usuário, e retornar mensagens "claras" quanto às ações necessárias. É o primeiro processo a ser disparado no nível do usuário, e dispara os demais processos, aceitando inclusive passagem de parâmetros para eles.

b) Do Ponto de Vista do Sistema

Um sistema operacional é um software orientado a eventos, ou seja, enquanto não há eventos a tratar ele fica simplesmente ocioso.

Na ocorrência de uni evento salva-se o estado da CPU e o de qualquer processo que porventura esteja em execução, determiná-se qual o tipo do evento e executa-se os programas necessários. Eventos podem ser:

- Interrupções cor hardware: quando um periférico termina um serviço é ele quem interrompe o processo em execução na CPU, para avisar que está livre. Uma vez iniciado um serviço de E/S a CPU pode esperar que ele termine ou prosseguir com a execução. Para esperar podemos simplesmente usar o estado especial WAIT ou ficar num loop ocioso do tipo idle: JMP idle; Para prosseguir o S.O. precisa manter tabelas com informações sobre quem solicitou serviço a qual periférico, se está ou não em execução, etc... conhecidos como Device Status Tables ou Request Queues.

- Chamadas ao sistema para: solicitação de recursos (memória e dispositivos), solicitação de E/S, solicitação de informações (data, hora, etc), disparo de processos, término normal/anormal de processos.

- "Traps": são tipos especiais de interrupções, geradas internamente à CPU, devido a tentativas de execução de instruções ilegais ou privilegiadas, refe-

rências ilegais à memória, tentativa de divisão por zero, erro de paridade, etc.

1.5 Características dos Sistemas Operacionais

A principal característica de um Sistema Operacional, independente de seu tipo (batch, interativo, tempo real), é a sua estrutura. Na análise destas características é importante distinguir claramente os mecanismos do SO, (determinam como fazer) das políticas do S.O. (determinam o que fazer).

1.5.1 Sistemas com Estrutura Monolítica

São sistemas antigos, desestruturados, formados por uma coleção de procedimentos (os vários módulos são compilados e linkados juntos), onde qualquer um pode chamar (pedir serviço) aos demais. A dificuldade de manutenção e o risco de loop de chamadas são as principais desvantagens. Mesmo com interfaces bem definidas quanto aos parâmetros e resultados, e com mecanismos de interrupção vetorizados não apresentam desempenho satisfatório.

1.5.2 Sistemas com Estrutura em Camadas

Nestes sistemas definimos vários níveis de serviço, ou camadas, sendo que uma camada só pode pedir serviço à inferior e prestar serviço à superior, o que facilita a manutenção e evita o risco de loop de chamada.

Exemplo 1: Sistema THE, de Dijkstra.

- nível 5: Programas do usuário,
- nível 4: "Buffering" de dispositivos de E/S
- nível 3: "Driver" do console do operador
- nível 2: Gerente de memória
- nível 1: Escalonador de CPU; operações "P" e "V"
- nível 0: Hardware

Exemplo 2: Sistema VENUS, de Liskov.

- nível 6: Programas do usuário
- nível 5: "Device drivers" e escalonadores
- nível 4: Memória virtual

- nível 3: Canais de E/S

- nível 2: Escalonador de CPU; operações "P" e "V"

- nível 1: Interpretador de instruções

- nível 0: Hardware

1.5.3 Sistemas com Estrutura de Máquinas Virtuais (VM da IBM)

Usando um eficiente escalonador para a CPU e técnicas de memória virtual este sistema cria a ilusão de um processador para cada processo, como se estes estivessem usando o Hardware real "puro", sem nenhum S.O. "Em cima" desse hardware virtual pode-se executar qualquer software que o reconheça, mas tipicamente é usado o CMS (Conversational Monitor System), um monitor rnono-usuário, interativo, que permite dois modos de operação: modo monitor e modo usuário virtual.

1.5.4 Sistemas com Estrutura Cliente/Servidor

Se considerarmos que estruturas "simples" como os sistemas VM implicam na verdade em programas muito complexos, chegaremos à solução inversa, ou seja, moveremos esta complexidade de código "para cima", mantendo um núcleo mínimo e migrando as funções do sistema para o nível do usuário.

Os processos *do* usuário (processos clientes) enviam solicitações de serviços através do núcleo, a processos servidores. Nesta situação o núcleo é quase que somente um servidor de mensagens. Os servidores são simples, pois são dedicados, o que facilita a sua manutenção, e como são executados em modo usuário os "bugs" não derrubam todo o sistema. Uma outra vantagem é que este modelo é facilmente adaptável a sistemas de computação distribuídos.

Algumas funções do S.O. continuam precisando ser executadas em modo supervisor, ou seja, alguns servidores,devem "rodar" em modo protegido. Neste caso o núcleo fornece os mecanismos, e as políticas continuam no nível do usuário. Por exemplo: em uma máquina que use E/S mapeado em memória, enviar uma mensagem para um certo endereço pode signmcar escrita absoluta na porta de acesso ao disco. Como o núcleo não verifica (não é sua função) o conteúdo das mensagens devemos usar algum mecanismo que só permita isso aos processos autorizados.

1.6 Exemplos de Sistema Operacional

1.6.1 MS-DOS

A sigla MS-DOS significa Microsoft Disk Operating System que em português significa Sistema Operacional de Disco. O prefixo MS representa Microsoft, empresa que criou o sistema. Este programa foi desenvolvido para permitir ao usuário realizar todas as funções básicas e essenciais necessárias no computador.

O MS-DOS é o Sistema Operacional mais utilizado e faz parte do Software Básico (programa indispensável ao funcionamento do computador). É um programa que se encarrega do Hardware do computador, por isso que é muito especial. Com pouquíssimas exceções, qualquer outro programa que é executado em seu computador é executado com a ajuda do DOS, em outras palavras o DOS é o programa que gerencia os componentes básicos do computador e os aloca a seus programas quando necessário. O DOS fica sob seu controle e existe para fornecer-lhe uma forma de comunicar suas instruções ao computador. Você informa instruções ao DOS através de comandos que ele reconhecerá. A maior parte desses comandos consistem em palavras baseadas na língua inglesa, pôr exemplo: copy, rename, date,time, label etc.

1.6.2 UNIX

As raízes do UNIX datam de meados dos anos 60, quando a AT&T, Honeywell, GE e o MIT embarcaram em um massivo projeto para o desenvolvimento de um utilitário de informação, chamado Multics (Multiplexed Information and Computing Service).

Multics era um sistema modular montado em uma bancada de processadores, memórias e equipamentos de comunicação de alta velocidade. Pelo desenho, partes do computador poderiam ser desligadas para manutenção sem que outras partes ou usuários fossem afetados.

Em 1973 o UNIX foi reescrito em C, talvez o fato mais importante da história deste sistema operacional. Isto significava que o UNIX poderia ser portado para o novo hardware em meses, e que mudanças eram fáceis. A linguagem C foi projetada para o sistema operacional UNIX, e portanto há uma grande sinergia entre C e UNIX.

Em 1975 foi lançada a V6, que foi a primeira versão de UNIX amplamente disponível fora dos domínios do Bell Labs, especialmente em universidades. Este foi o início da diversidade e popularidade do UNIX. Nesta época a Universidade de Berkley comprou as fontes do UNIX e alunos começaram a fazer modificações ao sistema.

Surgiram outras versões com a inclusão de novas características. O 4.2 BSD foi talvez umas das mais importantes versões do UNIX. O seu software de conexão de redes tornava muito fácil a tarefa de conectar computadores UNIX a redes locais. Nessa versão é que foram integrados os softwares que implementam TCP/IP e sockets.

O 4.4 BSD foi lançado em 1992 para várias plataformas: HP 9000/300, Sparc,

386, DEC e outras, mas não em VAX. Entre as novas características estão:

- Novo sistema de memória virtual baseado em Mach 2.5
- Suporte ISO/OSI (baseado em ISODE)

A Sun Microsystem também lançou a sua versão do UNIX a partir do BSD. Isto ocorreu até a versão SunOs 4.x. A nova versão, SunOs 5.x está baseada no SVR4, embora tenha herdado algumas características do SunOs 4.x. O novo sistema operacional da Sun, Solaris 2.x, engloba SunOs 5.x, Open Network Computing e Open Windows. É o solaris que provê o pacote de compatibilidade entre os BSD/SunOs e o SVR4/SunOs 5.x.

A Microsoft também lançou uma versão do UNIX, chamada XENIX, que rodava em PCs. Este sistema era inicialmente baseado na Versão 7, depois herdou características dos SIII e depois do SV.

1.6.3 WINDOWS NT

O Microsoft Windows NT começou a surgir em 18 de setembro de 1996, quando a Intel Corporation e a Microsoft Corporation anunciaram que estavam trabalhando no desenvolvimento de um novo sistema operacional para a futura família de processadores de 64 bits da Intel. O Windows NT é o sistema operacional da próxima geração, visando operar PCs até boa parte do próximo século. Foi projetado para ser um sistema operacional portável, capaz de se adequar facilmente a diversas plataformas de hardware, incluindo ambientes de um só processador e de múltiplos processadores.

Ele poderá ser facilmente estendido ou aperfeiçoado conforme o hardware evoluir. Pôr se mover para uma implementação completa de 32 bits, deixou para trás muitos cacoetes e problemas associados aos sistemas mais antigos de 16 bits.

Uma meta primária do Windows NT foi a compatibilidade com outros sistemas operacionais para PCs e com os programas projetados para rodas sob eles. Ou seja, o Windows NT foi projetado para permitir compatibilidade regressiva com a grande base de aplicações para PC existentes.

O Windows NT também foi projetado para satisfazer ou exceder os padrões atuais de desempenho. Um outro aspecto importante é que ele pode rodar em computadores com múltiplas CPUs.

1.6.4 WINDOWS 95

Criado pela Microsoft Corporation o Windows 95 é um software básico classificado na categoria de "Sistema Operacional". Ele cria uma interface gráfica para o usuário (GUI - Graphical User Interface) para proporcionar a este uma comunicação mais intuitiva e fácil com o computador. Este software usa a metáfora da mesa de trabalho (desktop) para dispor e arranjar informações gráficas e textuais na tela. O usuário tem acesso a essas informações através do mouse, que é usado para abrir janelas, selecionar opções, acionar vários objetos através de ícones, mover, copiar, renomear ou excluir arquivos, executar programas, etc.

O Windows 95 incorporou um conjunto de tecnologias que, somadas as inovações de sua interface, significam uma autêntica revolução no uso de micros. Uma das mudanças refere-se a própria interface gráfica, que evoluiu para facilitar ainda mais a maneira como o indivíduo se relaciona com o equipamento. Essa melhoria beneficia tanto usuários que conhecem pouco ou quase nada de microinformática quanto profissionais.

Outro avanço significativo é o suporte Plug-and-Play - ligue e use - automatizando totalmente a instalação e configuração da máquina.

Esta tecnologia acaba com os problemas de instalação de placas e outros periféricos, pois autoconfigura os componentes e põe fim aos conflitos de endereço e interrupção de memória.

O Windows 95 não traz apenas mudanças na interface. Sua grande mudança ocorreu nos bastidores. Ao contrário do Windows 3.1 (que é ambiente operacional), o Windows 95 é um sistema operacional integrado completo, que não trabalha "sobre" o MS-DOS. O Windows 95 elimina as limitações de memória herdadas do DOS.

O Windows 95 também tem uma vantagem em relação a outros sistemas e ambientes operacionais: ele permite a criação de nomes longos de arquivos, assim o usuário poderá gravar arquivos com nomes que realmente indiquem o que o arquivo representa. No Windows 95 os nomes de arquivos podem ter até 255 caracteres, incluindo espaços.

1.7 Principais Características dos Sistemas Operacionais

a) **Sistema monousuário** - Permite que apenas um usuário utilize o equipamento pôr vez (como o próprio nome diz: computador pessoal).

b) **monoprogramável** - Pôr possuir uma arquitetura simples, não necessita de rotinas de gerenciamento para compartilhamento de alguns recursos, tais como processador, arquivos, etc.

c) **Estrutura hierárquica dos dados** - Possibilita a organização dos arquivos em estrutura de diretórios e sub-diretórios permitindo uma melhor performance na utilização do equipamento.

d) **Redirecionamento de Entrada de Saída padrão** - Permite a modificação da entrada ou saída de periféricos padrão de alguns comandos para outros periféricos.

2 HISTÓRICO

Inicialmente, existiu somente o *hardware* do computador. Os primeiros computadores eram máquinas fisicamente muito grandes que funcionavam a partir de um console, que consiste em um periférico ou terminal que pode ser usado para controlar a máquina por métodos manuais, corrigir erros, determinar o estado dos circuitos internos e dos registradores e contadores, e examinar o conteúdo da memória. O console é o meio de comunicação entre o homem e a máquina, ou seja, é o meio por onde o operador fornece as entradas e por onde recebe as saídas. O console das primeiras máquinas consistia em chaves pelas quais o operador inseria informações, e por luzes indicativas das saídas, que podiam ser impressas ou perfuradas em uma fita de papel.

Com o passar do tempo, o uso de teclados para entrada de dados se tornou comum, e a saída passou a ser inicialmente impressa em papel. Posteriormente o console assumiu a forma de um terminal com teclado e vídeo.

Nesta época, os programadores que quisessem executar um programa, deveriam carregá-lo para a memória manualmente através de chaves no painel de controle, ou através de fita de papel ou cartões perfurados. Em seguida, botões especiais eram apertados para iniciar a execução do programa. Enquanto o programa rodava, o programador/operador podia monitorar a sua execução pelas luzes do console. Se erros eram descobertos, o programa precisava ser interrompido, e o programador podia examinar os conteúdos da memória e registradores, depurando-o diretamente do console. A saída era impressa diretamente, ou ainda perfurada em fita ou cartão para impressão posterior.

As dificuldades nesta época eram evidentes. O programador era também o operador do sistema de computação. Devido à escassez de recursos, a maioria dos sistemas usava um esquema de reserva para alocação de tempo da máquina. Se você quisesse usar o computador, deveria reservar um horário em uma planilha.

Além disso, este método não era eficiente na utilização de recursos. Supondo que você tivesse reservado 1 hora de tempo de computador para executar um programa em desenvolvimento. Se você tivesse alguns erros desagradáveis você provavelmente não terminaria dentro de 1 hora, e deveria juntar seus resultados e liberar a máquina para a próxima pessoa da fila. Por outro lado, se o seu programa rodasse sem problemas, você poderia terminar tudo em 35 minutos, e a máquina ficaria ociosa até a próxima reserva de horário.

Como as máquinas nesta época custavam muito dinheiro, pensou-se em algumas soluções para agilizar a tarefa de programação. Leitoras de cartões, impressoras de linha e fitas magnéticas tornaram-se equipamentos comuns. Montadores (*assemblers*), carregadores (*loaders*) e ligadores (*linkers*) foram projetados. Bibliotecas de funções comuns foram criadas para serem copiadas dentro de um novo programa sem a necessidade de serem reescritas.

Um bom exemplo do uso das bibliotecas de funções é sobre as rotinas que executavam operações de entrada e saída (E/S). Cada novo dispositivo tinha suas próprias características, necessitando de cuidadosa programação. Uma subrotina especial foi escrita para cada tipo de dispositivo de E/S. Essas subrotinas são chamadas de *device drivers* (controladores de dispositivos), e sabem como "conversar" com o dispositivo para o qual foram escritas. Uma tarefa simples como ler um caractere de um disco pode envolver seqüências complexas de operações específicas do dispositivo. Ao invés de escrever código a cada momento o *device driver* é simplesmente utilizado a partir de uma biblioteca.

Mais tarde, compiladores para linguagens de alto nível, como FORTRAN e COBOL, surgiram, facilitando muito a tarefa de programação, que antes era feita diretamente na linguagem da máquina. Entretanto a operação do computador para executar um programa em uma linguagem de alto nível era bem mais complexa.

Por exemplo, para executar um programa FORTRAN, o programador deveria primeiramente carregar o compilador FORTRAN para a memória. O compilador normalmente era armazenado em fita magnética, e portanto a fita correta deveria ser carregada para a unidade leitora de fitas magnéticas. Uma vez que o compilador estivesse pronto, o programa fonte em FORTRAN era lido através de uma leitora de cartões e escrito em outra fita. O compilador FORTRAN produzia saída em linguagem *assembly* que precisava ser montada (*assembled*), isto é, convertida para código de máquina. A saída do montador era ligada (*linked*) para suportar rotinas de biblioteca. Finalmente, o código objeto do programa estaria pronto para executar e seria carregado na memória e depurado diretamente no console, como anteriormente.

Podemos perceber que poderia existir um tempo significativo apenas para a preparação da execução de um *job* (tarefa). Vários passos deveriam ser seguidos, e em caso de erro em qualquer um deles, o processo deveria ser reiniciado após a solução do problema.

2.1 O Monitor Residente

O tempo de preparação de um *job* era um problema real. Durante o tempo em que fitas eram montadas ou o programador estava operando o console, a UCP (Unidade Central de Processamento) ficava ociosa. Vale lembrar que no passado muitos poucos computadores estavam disponíveis e eram muito caros (milhões de dólares). Além disso, os custos operacionais com energia, refrigeração, programadores, etc., tornava ainda mais cara sua manutenção. Por isso, tempo de processamento tinha muito valor, e os proprietários dos computadores os queriam ocupados o máximo do tempo possível. O computador precisava ter uma alta utilização para que o investimento fosse compensado.

Uma primeira solução foi contratar um profissional que operasse o computador. O programador não precisava mais operar a máquina, e assim que um *job* terminasse, o operador podia iniciar o próximo; não existia a ociosidade de tempo devido à reserva de tempo de computador não utilizada. Já que o operador tinha mais experiência com a montagem de fitas, o tempo de preparação foi reduzido. O usuário apenas fornecia cartões ou fitas perfuradas contendo o programa, e instruções necessárias para sua execução. Caso erros ocorressem durante a execução do programa, o

operador emitia uma listagem dos conteúdos da memória e registradores para que o programador pudesse depurar seu programa. Em seguida o próximo *job* era posto em execução, e assim por diante.

Além disso, para reduzir ainda mais o tempo de preparação, *jobs* com necessidades similares eram agrupados (*batched*) e executados em grupo pelo computador. Por exemplo, supondo que o operador tivesse recebido um *job* FORTRAN, um COBOL, e outro FORTRAN. Se ele os executasse nessa ordem, ele teria que preparar o *job* FORTRAN (carregar fitas de compilador, etc.), então o COBOL, e novamente o FORTRAN. Se ele executasse os dois *jobs* FORTRAN como um grupo ele prepararia o ambiente FORTRAN apenas uma vez economizando tempo de preparação.

Esta abordagem marcou uma época, onde o **processamento em batch** (lotes) definiu uma forma de utilização do computador: os usuários preparavam seus programas e dados, entregavam-nos ao operador do computador, que os agrupava segundo suas necessidades e os executava, produzindo as saídas para serem devolvidas aos respectivos programadores.

Mesmo assim, quando um *job* parava, o operador teria que notar o fato observando no console, determinar porque o programa parou (término normal ou anormal), listar conteúdos de memória se necessário e então carregar a leitora de cartões ou de fita de papel com o próximo *job* e inicializar o computador novamente. Durante a transição entre os *jobs*, novamente a UCP ficava ociosa.

Para resolver este problema, foi desenvolvido um seqüenciador automático de *jobs*, que consistia em um primeiro **sistema operacional** rudimentar. Sua função era controlar a transferência automática de um *job* para outro. Este programa foi implementado sob a forma de um **monitor residente**, sempre presente na memória da máquina para este fim.

Assim que o computador era ligado, o monitor residente era chamado, e transferia o controle para um programa. Quando o programa terminava, ele retornava o controle para o monitor residente, que ia para o próximo programa. Assim, o monitor residente fornecia uma seqüência automática entre programas e *jobs*.

Para que o monitor residente pudesse saber qual programa deveria ser executado e de que forma, cartões de controle foram introduzidos, de maneira muito semelhante às instruções que os operadores recebiam dos programadores para execução de seus programas. Assim, além do programas e dos dados para um *job*, cartões especiais de controle eram introduzidos entre os cartões de programa e dados do *job* a executar, como por exemplo:

$JOB - Primeiro cartão, indicando o início de um *job*;
$FTN - Executar o compilador FORTRAN;
$LOAD - Carregar o programa compilado;
$RUN - Executar o programa carregado;
$END - Fim do *job*.

Os cartões de início e fim de *job* eram geralmente utilizados para contabilizar o tempo de uso da máquina, para que seu tempo de processamento pudesse ser cobrado do usuário. Por isso, às vezes incluíam parâmetros indicando o usuário do *job*, nome do *job*, etc.

Para distinguir cartões de controle dos demais cartões era necessário identificá-los com um caractere ou um padrão especial no cartão. Em nosso exemplo, o símbolo do dólar ($) foi utilizado para este fim. A linguagem JCL (*Job Control Language*) da IBM usava duas barras (//) nas primeiras duas colunas. A figura 1.1 ilustra este cenário.

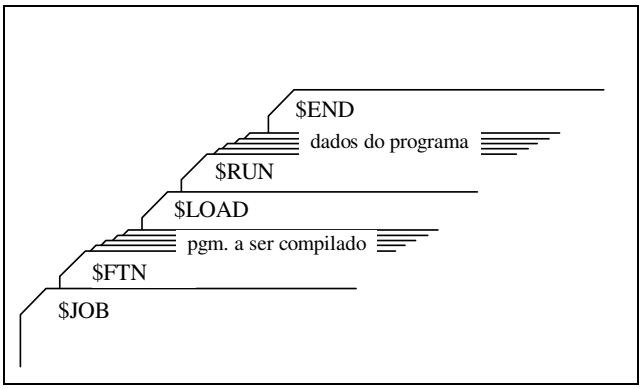


Figura 2.1 - *Deck* de cartões de um *job*

Um monitor residente tem várias partes identificáveis. Uma delas é o interpretador de cartões de controle, responsável pela leitura e extração das instruções dos cartões no instante da execução. O interpretador de cartões de controle chama um carregador em intervalos para carregar programas do sistema e programas de aplicação para a memória. Dessa forma, um carregador (*loader*) é uma parte do monitor residente. Ambos o interpretador de cartões de controle e o carregador precisam executar (E/S), assim o monitor residente tem um grupo de *drivers* de dispositivo para os dispositivos de do sistema. Freqüentemente, os programas de aplicação e do sistema estão ligados (*linked*) aos mesmos *drivers* de dispositivo, fornecendo continuidade na sua operação, bem como armazenando espaço de memória e tempo de programação. Um esquema de um monitor residente é mostrado na figura 1.2

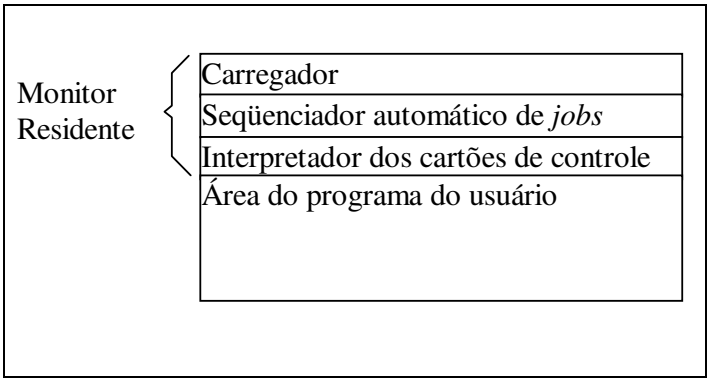


Figura 2.2 - Modelo de memória de um monitor residente

Sistemas *batch* utilizando este método funcionavam razoavelmente bem. O monitor residente fornece seqüenciamento automático dos *jobs* conforme a indicação dos cartões de controle. Quando um cartão de controle indica a execução de um programa, o monitor carrega o programa para a memória e transfere o controle para o mesmo. Quando o programa termina, ele retorna o controle para o monitor, que lê o próximo cartão de controle, carrega o programa apropriado e assim por diante. Este ciclo é repetido até que todos os cartões de controle sejam interpretados para o *job*. Então o monitor continua automaticamente com o próximo *job*.

2.2 Operação *Off-Line*

O uso de sistemas *batch* com seqüenciamento automático de *jobs* aumentou a performance do sistema. Entretanto, ainda assim a UCP ficava freqüentemente ociosa, devido à baixíssima velocidade dos dispositivos mecânicos em relação aos eletrônicos.

Os dispositivos de E/S mais lentos podem significar que a UCP fica freqüentemente esperando por E/S. Como um exemplo, um montador ou compilador pode ser capaz de processar 300 ou mais cartões por segundo. Uma leitora de cartões mais rápida, por outro lado, pode ser capaz de ler apenas 1200 cartões por minuto (20 cartões por segundo). Isto significa que montar um programa com 1200 cartões precisa de apenas 4 segundos de UCP, mas 60 segundos para ser lido. Dessa forma, a UCP fica ociosa por 56 dos 60 segundos, ou 93.3% do tempo. A utilização de UCP resultante é de apenas 6.7%. O processo é similar para operações de saída. O problema é que, enquanto uma operação de E/S está acontecendo, a UCP está ociosa, esperando que o E/S termine; enquanto a UCP está executando, os dispositivos de E/S estão ociosos.

Uma solução simples era substituir as lentas leitoras de cartão (dispositivos de entrada) e impressoras de linha (dispositivos de saída) por unidades de fita magnética. A maioria dos sistemas no final dos anos 50 e começo dos anos 60 eram sistemas *batch* cujos *jobs* eram lidos de leitoras de cartão e escritos em impressoras de linha ou perfuradoras de cartões. Ao invés de a UCP ler diretamente os cartões, os cartões eram primeiro copiados para uma fita magnética. Quando a fita estava suficientemente cheia ela era transportada para o computador.

Duas abordagens para esta solução (operação *off-line*) foram usadas. Dispositivos específicos (leitoras de cartão, impressoras de linha) foram desenvolvidos para provocar saída ou entrada direta de fitas magnéticas. A outra abordagem foi dedicar um pequeno computador para a tarefa de copiar de e para a fita. O pequeno computador foi um satélite do computador principal. Processamento satélite foi um dos primeiros casos de múltiplos sistemas de computação trabalhando em conjunto para aumentar a performance.

A principal vantagem da operação *off-line* foi de que o computador principal não estava mais restrito pela velocidade das leitoras de cartão e impressoras de linha, e sim pela velocidade das unidades de fita magnética mais rápidas. Essa técnica de usar fitas magnéticas para todo o E/S podia ser aplicada com qualquer unidade de equipamento de registro (leitoras e perfuradoras de cartão, leitoras e perfuradoras de fitas de papel, impressoras).

Além disso, nenhuma modificação precisa ser feita para adaptar programas de aplicação da operação direta para *off-line*. Considere um programa que executa em um sistema com uma leitora de cartões acoplada. Quando ele quer um cartão, ele chama o *driver* de dispositivo da leitora de cartão no monitor residente. Se a operação de cartão está em modo *off-line*, apenas o *driver* de dispositivo deve ser modificado. Quando um programa precisa de um cartão de entrada, ele chama a mesma rotina de sistema como antes. Entretanto, agora o código para aquela rotina não é o *driver* da leitora de cartões, mas uma chamada para o *driver* da fita magnética. O programa de aplicação recebe a mesma imagem do cartão em ambos os casos.

Essa habilidade de executar um programa com dispositivos de E/S diferentes é chamada **independência de dispositivo**. Independência de dispositivo torna-se possível pela existência de um sistema operacional que determina qual o dispositivo que um programa realmente usa quando faz o pedido de E/S. Programas são escritos para usar dispositivos de E/S lógicos. Cartões de controle (ou outros comandos) indicam como os dispositivos lógicos deveriam ser mapeados em dispositivos físicos.

O ganho real da operação *off-line* vem da possibilidade de usar múltiplos sistemas leitora-para-fita e fita-para-impressora para uma mesma UCP. Se a UCP pode processar com o dobro da velocidade da leitora, então duas leitoras trabalhando simultaneamente podem produzir fita suficiente para manter a UCP ocupada. Por outro lado, agora há um atraso mais longo para conseguir executar um *job* em particular. Ele deve ser lido antes para a fita. Existe o atraso até que *jobs* suficientes sejam lidos para a fita para preenchê-la. A fita deve ser então rebobinada, descarregada, manualmente carregada para a UCP e montada em um *drive* de fita livre. Além disso, *jobs* similares podem ser agrupados em uma fita antes de serem levados para o computador, fazendo com que às vezes um *job* tenha que “esperar” seu agrupamento com outros *jobs* similares em uma fita até que possa ser levado para a UCP.

2.3 Buferização

Processamento *off-line* permite a sobreposição de operações de UCP e E/S pela execução dessas duas ações em duas máquinas independentes. Se desejamos atingir tal sobreposição em uma única máquina, comandos devem ser colocados entre os dispositivos e a UCP para permitir uma separação similar de execução. Também, uma arquitetura adequada deve ser desenvolvida para permitir *buferização*.

Buferização é o método de sobrepôr E/S de um *job* com sua própria computação. A idéia é muito simples. Depois dos dados terem sido lidos e a UCP estar pronta para iniciar a operação, o dispositivo de entrada é instruído para iniciar a próxima entrada imediatamente. Dessa forma, a UCP e o dispositivo de entrada de dados ficam ambos ocupados. Com sorte, no instante em que a UCP está pronta para o próximo item de dado (registro), o dispositivo de entrada terá terminado de lê-lo. A UCP pode então começar o processamento dos novos dados lidos, enquanto o dispositivo de entrada começa a ler os dados seguintes. De forma semelhante isto pode ser feito para a saída. Nesse caso, a UCP cria dados que são colocados em um *buffer* até que o dispositivo de saída possa recebê-lo.

Na prática, é raro UCP e dispositivos de E/S estarem ocupados o tempo todo, já que ou a UCP ou o dispositivo de E/S termina primeiro. Se a UCP termina primeiro, ela deve esperar até que o próximo registro seja lido para a memória. É importante observar que a UCP não fica o tempo toda ociosa, no máximo o tempo que ficaria se não estivesse sendo utilizada buferização.

Por outro lado, se o dispositivo de entrada de dados termina primeiro, ele pode tanto esperar como continuar com a leitura do próximo registro. Neste caso ele só deverá parar quando os *buffers* estiverem cheios. Para que o dispositivo de entrada continue sempre trabalhando, normalmente os *buffers* costumam ter tamanho suficiente para mantê-lo sempre ocupado.

A buferização é geralmente uma função do sistema operacional. O monitor residente ou os *drivers* de dispositivo incluem *buffers* do sistema de E/S para cada dispositivo de E/S. Assim, chamadas ao *driver* de dispositivo pelos programas de aplicação normalmente causam apenas uma transferência do *buffer* para o sistema.
Apesar da buferização ser de alguma ajuda, ela raramente é suficiente para manter a UCP sempre ocupada, já que os dispositivos de E/S costumam ser muito lentos em relação à UCP.

2.4 Spooling

Com o passar do tempo, dispositivos baseados em discos tornaram-se comuns e facilitaram muito a operação *off-line* dos sistemas. A vantagem é que em um disco era possível escrever e ler a qualquer momento, enquanto que uma fita precisava ser escrita até o fim para então ser rebobinada e lida.

Em um sistema de disco, cartões são diretamente lidos da leitora de cartões para o disco. Quando um *job* é executado, o sistema operacional satisfaz seus pedidos por entrada da leitora de cartões pela leitura do disco. Da mesma forma, quando um *job* pede a impressão de uma linha para a impressora, esta é copiada em um *buffer* do sistema que é escrito para o disco. Quando a impressora fica disponível, a saída é realmente impressa.

Esta forma de processamento é chamada de *spooling* (spool = *Simultaneous Peripheral Operation On-Line*). *Spooling* utiliza um disco como um *buffer* muito grande para ler tanto quanto possa dos dispositivos de entrada e para armazenar arquivos de saída até que os dispositivos de saída estejam aptos para recebê-los. Esta técnica é também muito utilizada para comunicação com dispositivos remotos. A UCP envia os dados através dos canais de comunicação para uma impressora remota (ou aceita um *job* completo de entrada de uma leitora de cartões remota). O processamento remoto é feito em sua própria velocidade sem a intervenção da UCP. A UCP apenas precisa ser notificada quando o processamento termina, para que possa passar para o próximo conjunto de dados do *spool*.

A diferença entre buferização e *spooling* é que enquanto a buferização sobrepõe o processamento de um *job* com seu próprio E/S, o *spooling* sobrepõe o E/S de um *job* com o processamento de outros *jobs*. Assim, a técnica *spooling* é mais vantajosa do que a buferização. O único efeito colateral é a necessidade de algum espaço em disco para o *spool*, além de algumas tabelas em memória.

Outra vantagem do *spooling* vem do fato de que a técnica fornece uma estrutura de dados muito importante, que é a lista de *jobs*. A performance pode ser aumentada, pois os vários *jobs* armazenados no disco podem ser processados em qualquer ordem que o sistema operacional decidir, buscando o aumento de utilização da UCP (escalonamento de *jobs*). Quando *jobs* são lidos diretamente de cartões ou fita magnética, não é possível executar os *jobs* fora de ordem.

2.5 Multiprogramação

O aspecto mais importante do escalonamento de *jobs* é a habilidade de **multiprogramação**. As técnicas de operação *off-line*, buferização e *spooling* têm suas limitações na sobreposição de E/S. Em geral, um único usuário não pode manter tanto UCP e dispositivos de E/S ocupados o tempo todo. A multiprogramação aumenta a utilização de UCP, pois organiza os vários *jobs* de forma que a UCP sempre tenha algo para processar.

A multiprogramação funciona da seguinte maneira: inicialmente o sistema operacional escolhe um dos *jobs* da lista de *jobs* e começa a executá-lo. Eventualmente, o *job* deve esperar por alguma tarefa, como a montagem de uma fita, um comando digitado pelo teclado, ou mesmo o término de uma operação de E/S. Em um sistema **monoprogramado** a UCP permaneceria ociosa. Por outro lado, em um sistema **multiprogramado**, o sistema operacional simplesmente troca e executa outro *job*. Quando este novo *job* precisa esperar, a UCP troca para outro *job* e assim por diante. Em um dado momento, o primeiro *job* não precisa mais esperar e ganha a UCP. Assim, sempre que existirem *jobs* a serem processados, a UCP não ficará ociosa.

Sistemas operacionais multiprogramados são bastante sofisticados. Para que vários *jobs* estejam prontos para executar, é necessário que todos estejam presentes na memória RAM da máquina simultaneamente. Isto acarreta em um gerenciamento de memória para os vários *jobs*. Além disso, se vários *jobs* estão prontos para executar ao mesmo tempo, o sistema deve escolher qual deles deve ser executado primeiro. A política de decisão de qual *job* será executado é chamada de **escalonamento** de UCP. Por fim, o sistema operacional deve garantir que vários *jobs* rodando concorrentemente não afetem uns aos outros em todas as fases do sistema operacional, incluindo escalonamento de processos, armazenamento de disco e gerenciamento de memória.

2.6 Tempo Compartilhado

O conceito de sistemas de **tempo compartilhado**, também chamados de **multitarefa**, é uma extensão lógica de multiprogramação. Neste ambiente, múltiplos *jobs* são executados simultaneamente, sendo que a UCP atende cada *job* por um pequeno tempo, um a um em seqüência. Os tempos dedicados para cada *job* são pequenos o suficiente para que os usuários consigam interagir com cada programa sem que percebam que existem outros programas rodando. Quando muitos programas estão sendo executados, a impressão que o usuário tem é de que o computador está lento, pois a UCP tem mais *jobs* para atender, e portanto aumenta o tempo entre os sucessivos atendimentos para um determinado *job*.

É fácil de entender como funcionam sistemas de tempo compartilhado quando comparados com sistemas *batch*. Neste tipo de sistema operacional, um fluxo de *jobs* separados é lido (de uma leitora de cartões, por exemplo), incluindo seus cartões de controle que predefinem o que faz o *job*. Quando o *job* termina, seu resultado normalmente é impresso, e o próximo *job* é posto em execução.

A principal característica (e desvantagem) deste sistema é a **falta de interação** entre o usuário e o programa em execução no *job*. O usuário precisa entregar ao operador o programa que ele deseja executar, incluindo seus dados de entrada. Algum tempo depois (podendo demorar minutos, horas ou mesmo dias), a saída do *job* é retornada. Este tempo entre a submissão do *job* e seu término, chamado de tempo de *turnaround*, vai depender da quantidade de processamento necessária, tempo de preparação necessário, e da quantidade de *jobs* que estavam na fila antes dele ser submetido ao processamento.

Existem algumas dificuldades com o sistema *batch* do ponto de vista do programador ou do usuário. Já que o usuário não pode interagir com o *job* que está executando, o usuário deve indicar os cartões de controle para manipularem todos os resultados possíveis. Em um *job* de múltiplos passos, passos subseqüentes podem depender do resultado dos anteriores. A execução de um programa, por exemplo, pode depender do sucesso da compilação. Pode ser difícil definir completamente o que fazer em todos os casos.

Outra dificuldade em um sistema *batch* é que programas devem ser depurados estaticamente, a partir de uma listagem. Um programador não pode modificar um programa quando ele está sendo executado para estudar o seu comportamento, como hoje é possível na maioria dos ambientes de programação.

Um sistema de computação **interativo** (chamado de *hands-on*), fornece comunicação *on-line* entre o usuário e o sistema. O usuário dá instruções ao sistema operacional ou para um programa diretamente, e recebe uma resposta imediata. Usualmente, um teclado é usado para a entrada de dados e uma impressora ou monitor de vídeo para a saída de dados. Este tipo de **terminal** só apareceu algum tempo depois, com o barateamento de componentes eletrônicos neles utilizados. Quanto o sistema operacional termina a execução de um comando, ele passa a aceitar outros comandos do teclado do usuário, e não mais de uma leitora de cartões. Assim o usuário fornece o comando, espera pela resposta e decide o próximo comando, baseado no resultado do comando anterior. O usuário pode fazer experimentos com facilidade e pode ver resultados imediatamente.

Sistemas *batch* são bastante apropriados para executar *jobs* grandes que precisam de pouca interação. O usuário pode submeter *jobs* e retornar mais tarde para buscar os resultados; não é necessário esperar seu processamento. Por outro lado, *jobs* interativos costumam ser compostos por várias ações pequenas, onde os resultados de cada ação podem ser imprevisíveis. O usuário submete o comando e espera pelos resultados. O **tempo de resposta** deve ser pequeno - da ordem de segundos no máximo. Um sistema interativo é usado quando é necessário um tempo de resposta pequeno.

Conforme já vimos, no início dos tempos da computação, apesar de primitivos, os sistemas eram interativos. Um novo processamento só começava após o operador analisar os resultados do *job* anterior e decidir que ação tomar. Para aumentar o uso de UCP, sistemas *batch* foram introduzidos, o que realmente fez com que os computadores ficassem menos tempo ociosos. Entretanto, não havia interatividade nenhuma do usuário ou programador com o sistema.

Sistemas de tempo compartilhado foram desenvolvidos para fornecer o uso interativo de um sistema de computação a custos razoáveis. Um **sistema operacional de tempo compartilhado** (*time-sharing*) usa **escalonamento de UCP** e **multiprogramação** para fornecer a cada usuário uma pequena porção de tempo de computador.

Um sistema operacional de tempo compartilhado permite que muitos usuários **compartilhem** o computador simultaneamente. Já que cada ação ou comando em um sistema de tempo compartilhado tende a ser pequena, apenas uma pequena quantidade de tempo de UCP é necessária para cada usuário. Conforme o sistema troca de um usuário para outro, cada usuário tem a impressão de ter seu próprio computador, enquanto na realidade um computador está sendo compartilhado entre muitos usuários.

A idéia de tempo compartilhado foi demonstrada no início de 1960, mas já que sistemas de tempo compartilhado são mais difíceis e custosos para construir (devido aos numerosos dispositivos de E/S necessários), eles somente tornaram-se comuns até o início dos anos 70. Conforme a popularidade destes sistemas cresceu, os pesquisadores tentaram combinar os recursos de sistemas *batch* e de tempo compartilhado em um único sistema operacional. Muitos sistemas que foram inicialmente projetados como sistemas *batch* foram modificados para criar um subsistema de tempo compartilhado. Por exemplo, o sistema *batch* OS/360 da IBM foi modificado para suportar a opção de tempo compartilhado (*Time Sharing Option* - TSO). Ao mesmo tempo, à maioria dos sistemas de tempo compartilhado foi adicionado um subsistema *batch*. Hoje em dia, a maioria dos sistemas fornecem ambos processamento *batch* e de tempo compartilhado, embora seu projeto básico e uso sejam de um ou de outro tipo.

Sistemas operacionais de tempo compartilhado são sofisticados. Eles fornecem um mecanismo para execução concorrente. Como na multiprogramação, vários *jobs* deve ser mantidos simultaneamente na memória, o que requer alguma forma de gerenciamento de memória, proteção e escalonamento de UCP. Como a memória tem tamanho limitado, e em dadas situações alguns *jobs* terão que ser retirados da memória e gravados temporariamente em disco, para que outros programas possam ser lidos do disco e postos em execução na memória. Quando aquele *job* novamente precisar de continuação em sua execução, ele será trazido de volta para a memória.

Os primeiros sistemas operacionais para microcomputadores eram muito simples, pois o poder computacional dos primeiros “micros” era suficiente para atender somente a programas de um único usuário. Além do mais, os microcomputadores foram projetados na época para serem utilizados no máximo por uma pessoa em um determinado momento. Com o passar dos anos, os microcomputadores ganharam

poder de processamento equivalente a computadores que ocupavam salas inteiras no passado. Para aproveitar este potencial, os microcomputadores ganharam sistemas operacionais multitarefa, permitindo ao usuário executar mais de uma aplicação por vez, além de permitir situações desejáveis como imprimir um documento enquanto utilizando o editor de textos. Este processo se chama *swapping*, e para que isso possa ser feito, o sistema operacional deve fornecer gerenciamento de disco, e um sistema de arquivos *on-line*, além de proteção para que *jobs* não escrevam sobre *jobs* que foram colocados (*swapped*) em disco.

Hoje, multiprogramação e sistema compartilhado são os temas centrais dos sistemas operacionais modernos. Os sistemas operacionais mais recentes para microcomputadores suportam múltiplos usuários e múltiplos programas rodando concorrentemente em tempo compartilhado. Os sistemas mais conhecidos com essas características incluem: todas as versões de UNIX para PCs (UnixWare, SCO Unix, Linux, FreeBSD, Xenix, etc); o Microsoft Windows 3.x, Windows NT, e Windows 95; e o IBM OS/2. Apesar de pouco conhecidos, existem ainda alguns sistemas operacionais para PCs que rodam programas feitos para o MS-DOS, mas são multiusuário e multitarefa, como por exemplo o VM/386 e o VirtuOS/386 (produzido por uma empresa brasileira, a Microbase).

3 SISTEMAS DE ENTRADA E SAÍDA

Na figura 3.1, apresenta-se o diagrama simplificado de um sistema de computação.

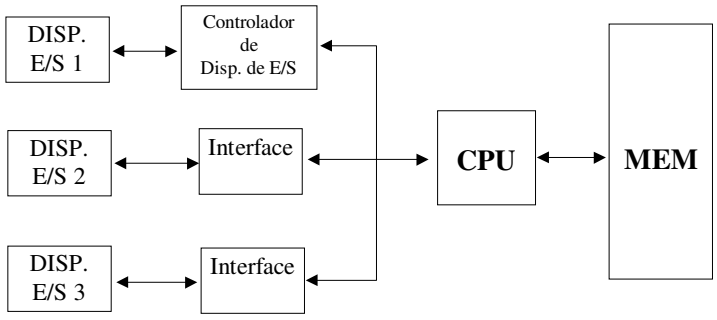


Figura 3.1. Diagrama Simplificado de um Sistema de Computação

As unidades de E/S são os dispositivos a partir dos quais a CPU realiza a interface com o mundo exterior. São considerados também unidades de E/S os dispositivos de armazenamento de informação auxiliares, tais como discos rígidos e flexíveis, unidades de fita magnética, etc.
É inquestionável a importância do sistema de E/S nos sistemas de computação. Sem os quais seria impossível enviar dados para processamento ou obter resultados de uma computação.

Sua importância também relaciona-se à velocidade dos sistemas de computação. Como só é possível acessar "perifericamente" a informação nos computadores, é importante que tais unidades de E/S sejam projetadas de tal forma a responder a altura da velocidade da CPU. Caso contrário pode-se ter uma CPU extremamente veloz, mas cujo acesso aos dados é limitado pelo desempenho do sistema de entrada e saída.

A CPU raramente acessa os periféricos diretamente, por razões de incompatibilidade elétrica, velocidade, formato de dados, etc. Normalmente existem circuitos que realizam as tarefas de interface e controle, no caso de periféricos mais complexos. A CPU por sua vez acessa estes circuitos para programá-los e para ler e escrever dados.

3.1 Mapeamento de Entrada e Saída

3.1.1 E/S Mapeada em Memória

Nos sistemas que empregam E/S mapeada em memória existe um espaço único de endereçamento.

O espaço destinado a E/S deverá ser implementado em certos endereços de memória. Tais endereços serão definidos pelo projetista de hardware do sistema. Nestes endereços estarão ligados os dispositivos de E/S e não posições de memória.

Observa-se que o acesso (leitura/escrita de dados) aos dispositivos de E/S será realizado por instruções de acesso à memória. O que distingüirá se o acesso é à memória ou à E/S será exclusivamente o endereço da posição de memória em questão. Na figura 3.2 apresenta-se um esquema simplificado de um sistema com E/S mapeada em memória.

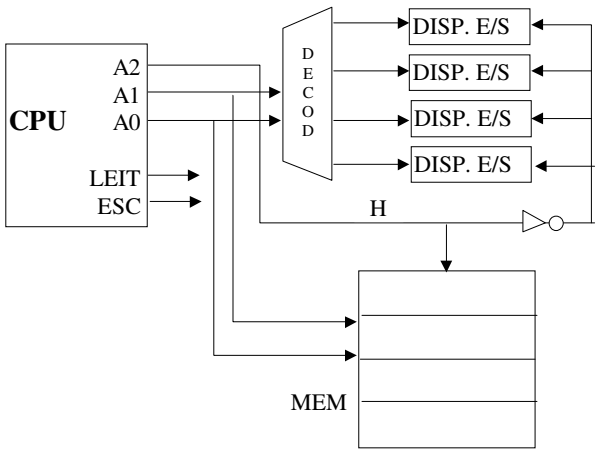


Figura 3.2. Sistema com E/S Mapeada em Memória

Existe um barramento único de 3 bits para endereçamento. Decidiu-se que metade do espaço total de armazenamento será destinado à área de memória e a outra metade a portas de E/S. Cada unidade (a de memória e a de E/S) receberá os dois bits menos significativos do endereço (A0 e A1). O bit mais significativo funcionará como um seletor (A2). Quando seu valor for igual a 1 o módulo de memória estará habilitado e o módulo de E/S desabilitado. Quando seu valor for igual a 0, o oposto.

3.1.2 E/S Mapeada em Espaço de E/S

Neste caso existem dois espaços de endereçamento: um dedicado à memória e outro dedicado às unidades de E/S. Não há necessidade de especificar-se o espaço de E/S como parte dos endereços de memória.

O espaço de E/S é acessado a partir de instruções específicas incorporadas ao conjunto de instruções da CPU.

Pode-se detectar uma CPU que apresenta E/S mapeada em espaço de E/S através da observação do barramento de controle. Estas CPUs apresentam sinais (pinos) especiais para indicar se o endereço gerado refere-se à memória ou às unidades de E/S, bem como instruções de máquina específicas para acesso à memória e acesso a posições de E/S. Quando a CPU executa uma instrução de acesso à memória ou à E/S tais pinos são correspondentemente ativados.

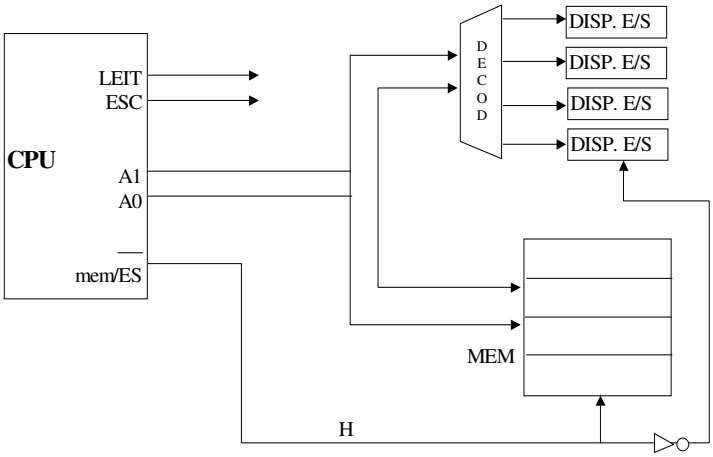


Figura 3.3. Sistema com E/S Mapeada em Espaço de E/S

Na figura 3.3, aparece um esquema simplificado de um sistema com E/S mapeada em espaço de E/S. O barramento é único para endereçamento da memória e de E/S como no caso anterior. Entretanto a especificação deste endereço (se é para memória ou para E/S) é feita pelo pino do barramento de controle MEM/~E.S.

Quando o valor lógico neste pino é igual a 1 isto indica que o endereço destina-se à memória, quando é igual a 0 para a E/S. Quando a CPU executa uma instrução de acesso à memória o circuito interno da CPU automaticamente escreve 1 no pino MEM/~E.S. e este sinal vai habilitar o banco de memória, e desabilitar as portas de E/S. Quando a instrução a ser executada é de E/S a situação oposta acontece.

Observações importantes:

1) A utilização de E/S mapeada em memória permite o uso de modos de endereçamento mais poderosos para acesso à E/S.
2) No entanto o código é menos claro para análise, já que a única diferença entre uma instrução de acesso à memória ou à E/S é o endereço da posição. No caso de E/S mapeada em espaço de E/S esta diferença aparace claramente no código como instruções diferentes para acesso à memória e para acesso à E/S.
3) Normalmente, instruções de acesso à memória têm um tempo de execução (número de ciclos de máquina) superior às instruções dedicadas de acesso à E/S.

4) As instruções dedicadas à transferência de E/S são normalmente simples (1 modo de endereçamento simples). Ex.: *IN* e *OUT.*

3.2 Métodos de Transferência Controlada por Programa

Existem várias formas de tramsferência de informação sob controle da CPU (de um programa de trsnaferência executado pela CPU).
Nesta seção detalharemos as 4 principais formas:

- Modo Bloqueado (*Busywait)*
- *Polling* (Inquisição)
- Interjeição
- Interrupção

3.2.1 Modo Bloqueado (Busywait)

No modo bloqueado, como o próprio nome indica, a CPU fica totalmente dedicada ao periférico do início ao fim de toda a operação de E/S. Ou seja, quando da execução de uma operação de E/S a CPU aguarda que o periférico a termine para prosseguir com o andamento do programa. Evidentemente neste modo a CPU é subutiluizada pois fica inativa durante a transferência, considerando-se que os periféricos são tipicamente lentos se comparados à CPU.

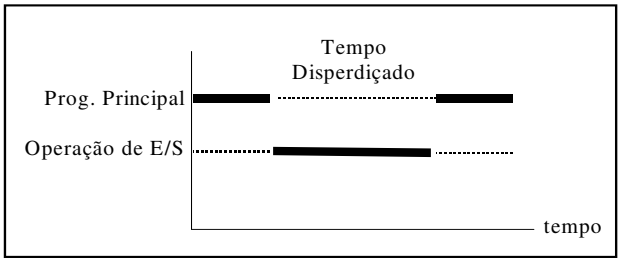


Figura 3.4. Gráfico de Uso da CPU Durante o Modo Bloqueado

3.2.2 Polling (Inquisição)

Nos Sistema com *Polling* (ou teste de estado) existe associado a cada dispositivo de E/S um flag (um flip-flop). A CPU testa periodicamente o conteúdo destes flags. Caso o conteúdo armazenado seja igual a 1 isto indica que o dispositivo associado necessita da atenção da CPU para realizar uma operação de E/S ou para completar/concluir uma operação de E/S ou para completar/concluir uma operação anteriormente iniciada.

Desta forma evita-se aquele tempo que a CPU fica bloqueada aguardando que o periférico termine alguma operação conforme descrito no ítem anterior (modo bloqueado).

No entanto, deve-se incluir uma rotina de teste dos flags associados aos dispositivos bem como os flags propriamente ditos. A rotina de teste, evidentemente, consumirá certo tempo do processamento da CPU, no entanto este será certamente inferior às somas dos tempos de espera do modo bloqueado.

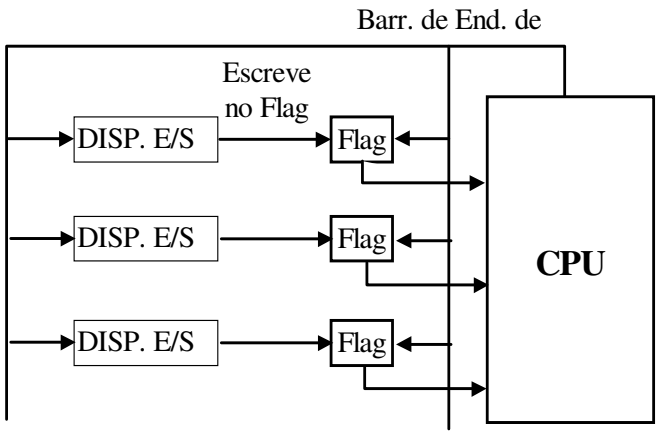


Figura 3.5. Esquema simplificado de Sistema com Mecanismo de Polling

Na Figura 3.5 mostramos o diagrama simplificado de um sistema com mecanismo de *Polling* para teste de estado dos periféricos. Nestes sistemas, a CPU interrompe periódicamente a execução de outras tarefas (programas) para executar a rotina de teste de flags.

A rotina testa então, cada um dos flags verificando o valor armazenado ali. Se o valor for igual a 1, o periférico associado necessita a atenção da CPU que chama a rotina de atendimento daquela dispositivo. Observe que a ordem de teste dos flags deixa implícito, nesta política, o estabelecimento de prioridades.

Em alguns casos pode-se optar por seguir a rotina de teste de flags após o atendimento de uma solicitação, em outros, a rotina é interrompida após atender-se a um dispositivo, vindo a ser ativada posteriormente. Quando este esquema é adotado, pode haver postergação infinita no atendimento de dispositivos de prioridade inferior, caso os primeiros dispositivos a serem testados requisitem com muita freqüência a atenção da CPU.

Caso, após a rotina de atendimento, testa-se os demais flags na seqüência, isto pode significar tempo gasto (não desprezível se o número de periféricos for muito grande). O projetista do sistema deverá optar entre uma destas soluções levando em conta:

a) número de periféricos
b) freqüência estimada de pedidos de atendimento por parte dos periféricos
c) número de pontos de teste de flags (chamada da rotina de *polling*) desejáveis dentro do programa
d) tempo de atendimento médio dos periféricos

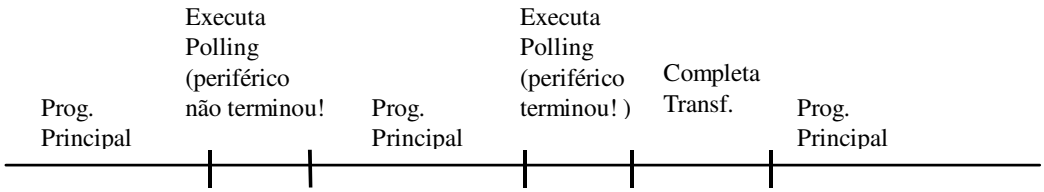


Figura 3.6. Execução Típica de um Sistema que Executa Polling para E/S

Observação: a prioridade no atendimento aos pedidos é dada pela ordem de teste dos flags.

3.2.3 Interjeição

Interjeição é um sistema aprimorado do *Polling*. Neste caso, antes de realizar o teste em cada flag, a CPU testa um flag adicional que representa o OU-Lógico de todos os flags associados aos periféricos.

Portanto, a CPU só testará os flags para descobrir qual o periférico que deseja realizar a transferência, quando o flag adicional estiver em 1, significando que há pelo menos um dispositivo de E/S com seu flag igual a 1.

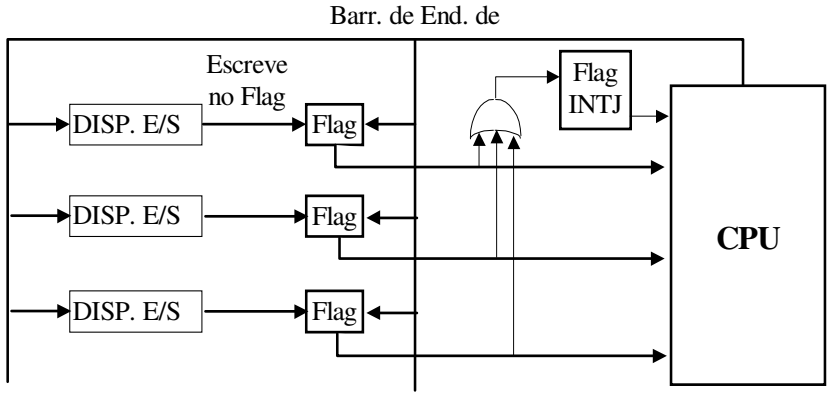


Figura 3.7. Esquema simplificado de Sistema com Mecanismo de Interjeição

3.2.4 Interrupção

Os métodos anteriores para transferência de informação, apesar de resolverem o problema e serem comparativamente mais baratos, apresentam sérios problemas em sistemas onde exige-se maior desempenho, tanto da CPU quanto dos periféricos. No modo bloqueado a CPU perde muito tempo esperando que o periférico conclua a operação para que o programa possa continuar sendo executado. Nos métodos de *polling* e interjeição, apesar de aprimorado, ainda exigem teste periódico da CPU para monitoração de seus estados.

A interrupção representa um avanço bastante grande em relação a estes métodos pois, neste caso, a CPU não necessita mais testar o estado dos periféricos. Desta forma não preocupa-se desnecessarimente com os periféricos. Na interrupção, quando um periférico necessita informar certa situação, continuar um procedimento, enviar uma informação anteriormente solicitada, enfim qualquer sinalização necessá-

ria à CPU, ele envia um sinal (interrupção) via uma linha especial (**INT**). Se a CPU estiver habilitada à responder a este pedido (a CPU pode estar desabilitada a atender INT's), a CPU "interrompe" a execução do programa e desvia a execução para a rotina de atendimento do periférico que enviou o sinal de **INT**. Concluída a rotina de atendimento, o programa que estava sendo executado antes da interrupção, volta a ser executado do ponto onde fora interrompido.

De uma forma mais explícita, interrupção pode ser definida da seguinte maneira:

1. Trata-se de um sinal gerado pelo hardware externo à CPU e dirigido à esta, indicando que um evento externo ocorreu e isto requer a atenção imediata da CPU;
2. Este sinal ocorre assincronamente a execução da seqüência de programa. A interrupção pode ocorrer a qualquer instante e isto não está sob controle do programa executado;
3. O hardware de controle das interrupções (interno à CPU), se as interrupções estiverem habilitadas, completa a execução da instrução corrente, e então força o controle a desviar para a rotina de atendimento do periférico que solicitou, e;
4. Uma instrução especial de "retorno-de-interrupção" é empregada para direcionar o controle de volta ao ponto do programa principal onde a interrupção ocorreu.

3.2.4.1 Interrupção c/ 1 Nível de Prioridade

Interrupção de um nível de prioridade referem-se a sistemas onde o programa uma vez interrompido, estando a máquina executando a rotina de atendimento do periférico solicitante, esta não pode ser interrompida. Somente após a conclusão da rotina de atendimento em execução é que o sistema poderá ser novamente interrompido.

Funcionamento: um sistema com interrupção de um nível de prioridade é esquematizado abaixo.

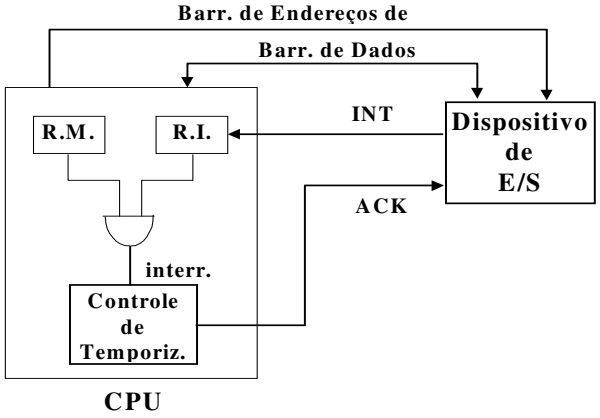


Figura 3.8. Sistema com Interrupção de um Nível de Prioridade.

R.M. = Reg. de Máscara
R.I. = Reg. de Interrupção
interr. = sinal de ocorrência de interrupção externo à CPU
INT = solicitação de interrupção
ACK (acknowledge) = reconhecimento da interrupção

Através do pino de **INT** o periférico que sinaliza a CPU a intenção de realizar uma operação de E/S. A CPU, se habilitada, responderá (após a conclusão do ciclo de instrução em andamento), som sinal de reconhecimento através da linha **ACK**.

Para realizar o controle do sistema de interrupção de um nível a CPU dispõe de estruturas ilustradas na figura 3.8.

O registrador **R.I.** armazena um pedido de interrupção e o registrador **R.M.** (registrador de máscara) determina se a CPU reconhecerá ou não a interrupção. Isto ocorre, pois o sinal de interrupção interno à CPU só irá ocorrer quando o conteúdo de **R.M.** for igual a 1. Na condição contrária, (**R.M.** = 0), havendo ou não pedido de interrupção, o hardware não habilitará o sinal de interrupção interno (**AND-**Lógico dos dois registradores).

Salvamento de Contexto:
Quando ocorre uma interrupção, o programa que está sendo executado pela CPU é interrompido, o sistema executa a rotina de atendimento e direciona o controle de volta ao programa principal.

Entretanto, para que a CPU seja capaz de retornar ao contexto anterior à interrupção, certas atividades devem ser executadas. Como o pedido de interrupção é totalmente assíncrono ao programa sendo executado, o pedido pode ocorrer no meio da execução de uma instrução. Neste caso, a CPU termina de executar a instrução corrente antes de autorizar o envio do sinal **ACK**. Para que possa voltar ao ponto exato onde parou no programa principal, a CPU deve guardar em alguma posição de memória (pilha) ou mesmo em um registrador apropriado o valor corrente no contador de programa (PC), ou seja, o endereço da próxima instrução a ser executada após a execução da rotina de atendimento da interrupção. Desta forma ao concluir a rotina de atendimento, a CPU atribui ao PC aquele valor armazenado e passa a executar novamente o programa principal no ponto correto.

Antes de enviar o sinal de **ACK**, a CPU escreve um 0 (zero) no registrador de máscara impedindo que ocorra qualquer interrupção. Quando a CPU volta a executar o programa principal, o registrador de máscara recebe 1, o que caracteriza a habilitação das interrupções.

Todos os procedimentos apresentados são executados automaticamente pelo hardware sem que o usuário interfira. Abaixo caracterizamos alguns dos passos executados pelo hardware de controle de interrupções:

- salvamento do PC
- desabilitação das int's

- envio do sinal de **ACK**
- retorno do PC anterior
- reabilitação das int's

A CPU pode dispor de instruções para habilitar e desabilitar interrupções. Seu uso fica a critério do programador. Tais instruções podem ser úteis para desabilitar as interrupções em trechos críticos de um programa.

Muitas vezes, mais informação deve ser salva para que quando o controle retorne da rotina de atendimento para o programa principal consiga restabelecer-se todo o contexto. Um caso típico são os registradores de uso geral e registrador de *status*. Caso o programador julgue que tal procedimento é necessário, este deve ser elaborado de forma explícita, dentro da rotina de atendimento.

3.2.4.2 Interrupção de um Nível de Prioridade com Vários Dispositivos

Quando temos vários dispositivos de E/S ligados a uma mesma linha de interrupção, antes de podermos executar a rotina de atendimento do dispositivo que solicitou atenção da CPU, é necessário descobrir que dispositivo fez a solicitação. No caso específico da figura I, a identificação será realizada através de uma rotina de software que, quando descobrir o dispositivo, chamará a rotina de atendimento correspondente.

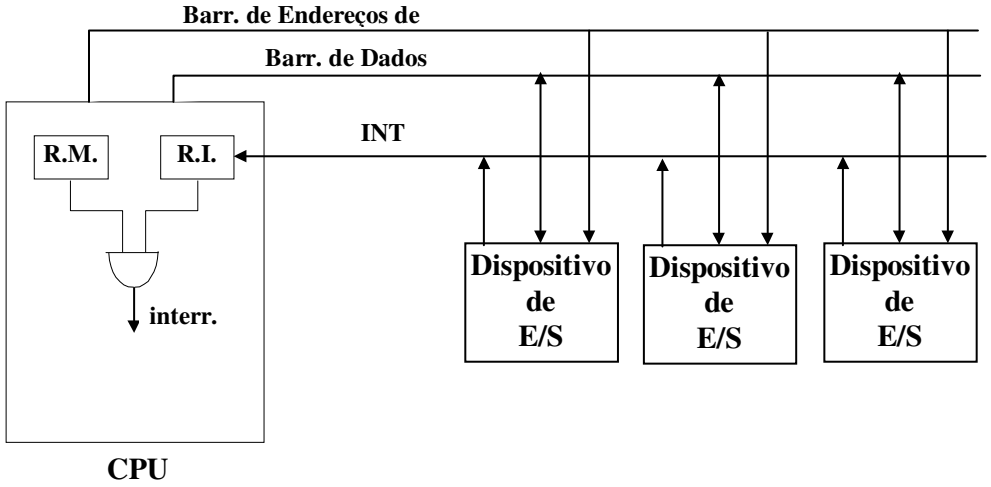


Figura 3.9. Sistema com um Nível de Prioridade e Vários Dispositivos

Observação: a prioridade no atendimento destes dispositivos (caso haja pedidos simultâneos) dependerá exclusivamente da rotina de identificação. Tal rotina será bastante semelhante a uma rotina de *polling*.

Uma outra forma (mais rápida) de identificar o dispositivo de mais alta prioridade que solicitou interrupção é implementar *polling* por hardware através de uma cadeia daisy-chain do sinal **ACK**, conforme inlustrado na figura abaixo.

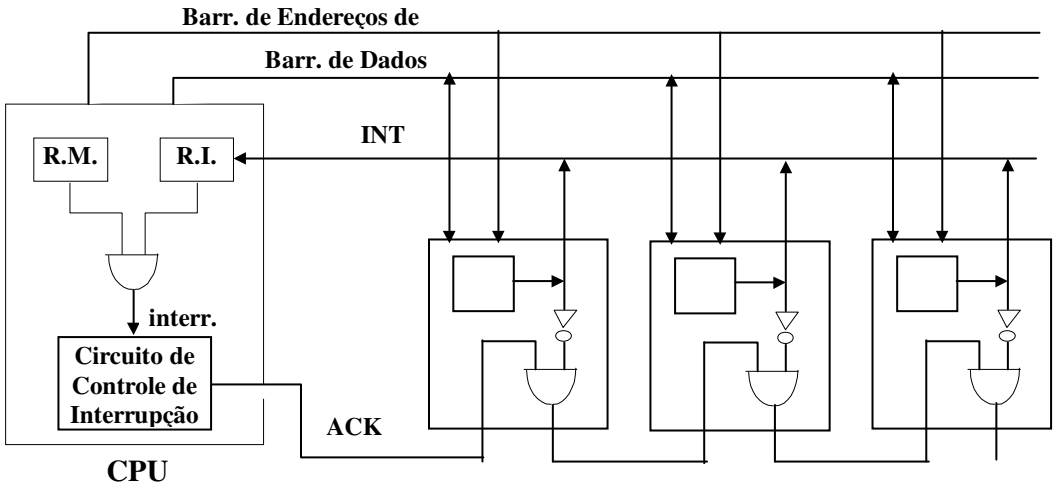


Figura 3.10. Cadeia daisy-chain de Propagação de Reconhecimento (**ACK**)

Neste caso quando a CPU aceita um pedido de interrupção e o contexto já foi automaticamente salvo, um sinal de reconhecimento é enviado através da linha **ACK**. Este sinal entra dentro da lógica dos dispositivos de E/S e propaga-se para os dispositivos seguintes até que atinja o primeiro dispositivo que tenha solicitado interrupção. A partir deste dispositivo o sinal propoagado será 0 (zero) indicando o não reconhecimento.

O dispositivo que pediu interrupção e recebeu o sinal de reconhecimento "segura" para si impedindo que os demais (na seqüência) o recebam. A seguir este dipositivo identifica-se para a CPU (escrevendo um código no barramento de dados, por exemplo) e esta direciona o controle para a rotina de atendimento do respectivo periférico. A prioridade é fixa pelo hardware, ou seja, a ordem segundo a qual os dispositivos estão dispostos em relação a linha de propagação do sinal **ACK**.

3.2.4.3 Interrupção c/ Múltiplos Níveis de Prioridade

A diferença efetiva entre um sistema com um nível de prioridade e outro com múltiplos níveis de prioridade é que este útlimo permite que a rotina de atendimento sendo executada (que foi chamada através de uma interrupção) pode ser interrompida por dispositivos que tenham "prioridade de atendimento"mais elevada, e assim sucessivamente.

Para possibilitar esta flexibilidade tais sistemas contam com outras características além daquelas existentes em sistemas com um único nível de prioridades.

Na figura abaixo, encontra-se esquematizado um sistema com múltiplos níveis de prioridade.

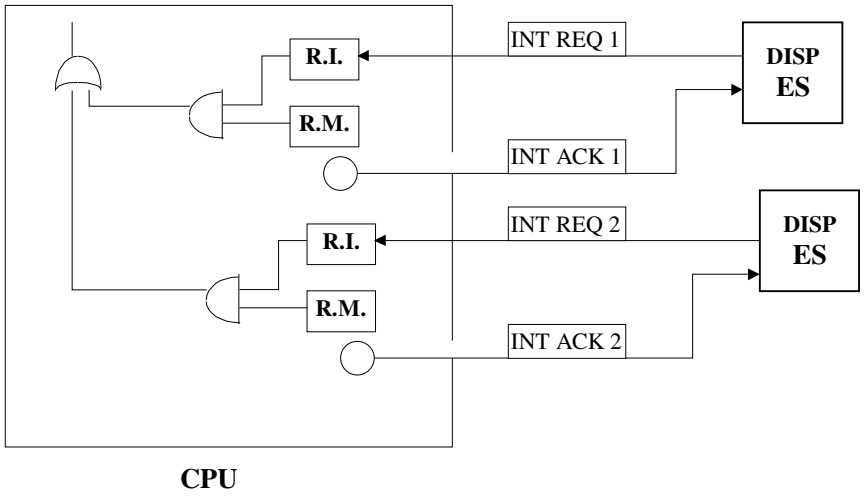


Figura 3.11. Múltiplos níveis de prioridade.

Associado a cada nível de interrupção (linha de pedido de int./ INT REQ) existe um flip-flop (R.I.) que armazena a requisição. Da mesma forma associado a cada registrador de interrupção existe um registrador de máscara que possui as mesmas atribuições que no caso com um nível de prioridade.

Quando acontece do AND-LÓGICO de qualquer par R.I./R.M. resultar em “1” a CPU é interrompida.

É claro que, para que a CPU suporte múltiplas interrupções aninhadas (a CPU é interrompida dentro da rotina de atendimento de uma outra interrupção), o hardware deve fornecer uma forma de retorno ordenado às rotinas das interrupções de níveis inferiores. Em outras palavras, o hardware da CPU deve guardar em ordem os PC’s das múltiplas ocorrências de forma que, a medida que as rotinas de n;ivel mais alto sejam terminadas, a execução se direcione para as rotinas de n;ivel mais baixo até o programa principal, na ordem inversa em que iniciaram. Isto normalmente é realizado em uma estrutura di tipo pilha que possui endereço inicial fixo.

Outro aspecto importante refere-se ao conteúdo de todos os registradores de máscara, ou, de uma forma simples à máscara de interrupção. Cada nível de interrupção deverá ter associado uma máscara. Esta máscara caracteriza os dispositivos (ou linhas de int.) que podem ou não interromper este nível.

Portanto, quando uma interrupção é reconhecida, a rotina de atendimento deve fornecer imediatamente a máscara de interrupção associada. Para que o contexto possa ser coerentemente retornado a máscara anterior deverá (antes de se escrever a nova) ser salva juntamente com o PC+1.

Desta forma, quando retorna-se de uma rotina de interrupção, tanto o antigo PC bem como a antiga máscara são restabelecidos. Quando o usuário pode definir as máscaras é preciso tomar cuidado com a coerência (interrupções de nível mais baixo não devem mascarar níveis mais altos e interrupções de nível mais alto não podem ser interrompidas por níveis mais baixos).

É evidente que múltiplas solicitações podem ocorrer simultaneamente e todas estarem habilitadas. O sistema deve prover uma forma de determinar qual delas é a de maior nível para chamar a respectiva rotina de atendimento. Isto pode ser feito através de uma rotina de software que varre os bits de solicitação não mascaradas ou por um circuito. Normalmente emprega-se a última solução por questões de desempenho.

As precauções para salvamento de contexto global continuam válidas para sistemas com múltiplos níveis de interrupção. Os procedimentos internos automáticos do tipo: aguardar o final da instrução em andamento para enviar o ACK etc. também continuam válidos e necessários.

3.2.4.4 Identificação da Fonte de Interrupção

Existem basicamente três formas de identificar-se a fonte de uma interrupção:

1. Endereço Fixo
2. Vetorada
3. Auto-vetorada

Endereço Fixo:
Neste método, quando a CPU recebe um pedido de interrupção o PC e a máscara são salvos e o PC é carregado sempre com um mesmo endereço. Isto faz com que a CPU seja desviada sempre para a mesma posição de memória. Nesta posição encontra-se a rotina de atendimento do dispositivo, ou, no caso de tratar-se de vários dispositivos, uma rotina de identificação do tipo polling, por exemplo.

Vetorada:
Neste método, quando o periférico recebe o sinal ACK, de reconhecimento do seu pedido de interrupção ele escreve uma palavra no barramento de dados. A CPU lê esta palavra e a utiliza como um índice para acessar uma tabela que contém os endereços de todas as rotinas de atendimento. Esta área, onde encontra-se a tabela, é normalmente pré-fixada e dedicada exclusivamente para esta finalidade.

Auto-vetorada:
O método de identificação auto-vetorsda, normalmente aparace associado ao método vetorado. Para ilustrar o funcionamento deste método, descreve-se o sistema de identificação de interrupção do 68000 que emprega também este método.

Os vetores de interrupção estão localizados em 1024 primeiros bytes da área de memória. Cada entrada no item possui 4 bytes que formam o endereço inicial da rotina de atendimento. Os vetores 25-31 são reservados para o modo auto-vetorado.

No modo auto-vetorado existem 7 níveis de prioridade de interrupção numerados de 1 a 7. O nível 7 é o de mais alta prioridade. Estas linhas são codificadas fornecendo uma entrada de três bits a CPU: as linhas IPL0. IPL1 e IPL2.
Estas linhas identificam diretamente o vetor de interrupção associado ao nível (25-31).

Quando as três linhas estão em zero isto indica que não existe pedido de interrupção. As linhas IPL0-IPL2 são, ao mesmo tempo:

- pedido de interrupção
- identificação da fonte

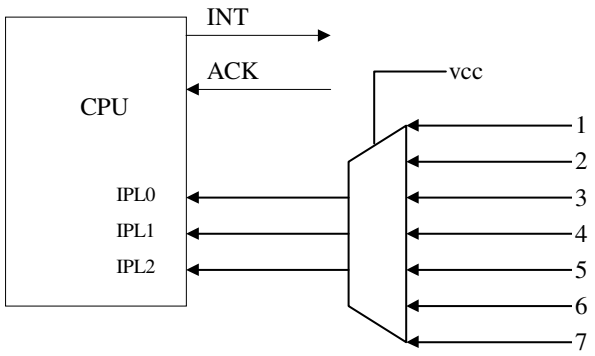


Figura 3.12. Sistema auto-vetorado do processador 68000

4 PROCESSOS

O conceito de processo é, certamente, o conceito mais importante no estudo de sistemas operacionais. Para facilitar o entendimento deste conceito, considere-se um computador funcionando em multiprogramação (isto é, tendo vários programas simultaneamente ativos na memória). Cada programa em execução corresponde a um procedimento (seqüência de instruções) e um conjunto de dados (variáveis utilizadas pelo programa). É conveniente ter-se instruções separadas dos dados, pois isso possibilita o compartilhamento do código do procedimento por vários programas em execução (neste caso diz-se que o procedimento e **reentrante** ou puro). Se cada programa em execução possui uma pilha própria, então os dados podem ser criados (alocados) na própria pilha do programa.

Além das instruções e dados, cada programa em execução possui uma área de memória correspondente para armazenar os valores dos registradores da UCP, quando o programa, por algum motivo, não estiver sendo executado. Essa área de memória é conhecida como **registro descritor** (ou bloco descritor, bloco de contexto, registro de estado, vetor de estado) e, além dos valores dos registradores da UCP, contém outras informações.

Assim, em um determinado sistema, cada programa em execução constitui um **processo**. Portanto, podemos definir processo como sendo um programa em execução, o qual é constituído por uma seqüência de instruções, um conjunto de dados e um registro descritor.

Num ambiente de multiprogramação, quando existe apenas um processador na instalação, cada processo é executado um pouco de cada vez, de forma intercalada. O sistema operacional aloca a UCP um pouco para cada processo, em uma ordem que não é previsível, em geral, pois depende de fatores externos aos processos, que variam no tempo (carga do sistema, por exemplo). Um processo após receber a UCP, só perde o controle da execução quando ocorre uma interrupção ou quando ele executa um *trap*, requerendo algum serviço do sistema operacional.

As interrupções são transparentes aos processos, pois o efeitos das mesmas é apenas parar, temporariamente, a execução de um processo, o qual continuará sendo executado, mais tarde, como se nada tivesse acontecido. Um *trap*, por outro lado, é completamente diferente, pois bloqueia o processo até que o serviço requerido pelo mesmo, ao sistema operacional, seja realizado.

Deve ser observado que um processo é uma entidade completamente definida por si só, cujas operações (instruções executadas) se desenvolvem no tempo, em uma ordem que é função exclusiva dos valores iniciais de suas variáveis e dos dados lidos durante a execução.

Em um sistema com multiprocessamento (com mais de uma UCP), a única diferença em relação ao ambiente monoprocessado é que o sistema operacional passa a dispôr de mais processadores para alocar os processos, e neste caso tem-se realmente a execução simultânea de vários processos.

Um sistema monoprocessado executando de forma intercalada N processos podem ser visto como se possuísse N **processadores virtuais**, um para cada processo em execução. Cada processador virtual teria 1/N da velocidade do processador real (desprezando-se o *overhead* existente na implementação da multiprogramação). O *overhead* de um sistema operacional é o tempo que o mesmo perde na execução de suas próprias funções, como por exemplo o tempo perdido para fazer a multiplexação da UCP entre os processos. É o tempo durante o qual o sistema não está produzindo trabalho útil para qualquer usuário.

Tanto no paralelismo físico (real, com várias UCP) como no lógico (virtual, uma UCP compartilhada), as velocidades relativas com que os processos acessarão dados compartilhados não podem ser previstas. Isto implica em mecanismos de sincronização entre processos, como vimos anteriormente com as instruções *parbegin/parend*.

Processos paralelos são denominados **concorrentes** ou **assíncronos** e, de acordo com o tipo de interação existente entre eles, podem ser classificados como **disjuntos** (não interativos), quando operam sobre conjuntos distintos de dados, ou **interativos**, quando têm acesso a dados comuns. Processos interativos podem ser **competitivos**, se competirem por recursos, e/ou **cooperantes**, se trocarem informações entre si.

No caso de computações realizadas por processos interativos, como a ordem das operações sobre as variáveis compartilhadas pode variar no tempo (pois as velocidades relativas dos processos dependem de fatores externos que variam no tempo), o resultado da computação pode não depender somente dos valores iniciais das variáveis e dos dados de entrada. Quando o resultado de uma computação varia de acordo com as velocidades relativas dos processos diz-se que existe uma **condição de corrida** (*race condition*). É necessário evitar condições de corrida para garantir que o resultado de uma computação não varie entre uma execução e outra. Condições de corrida resultam em computações paralelas errôneas, pois cada vez que o programa for executado (com os mesmos dados) resultados diferentes poderão ser obtidos. A programação de computações paralelas exige mecanismos de sincronização entre processos, e por isso sua programação e depuração são bem mais difícies do que em programas tradicionais.

A maioria das linguagens de programação existentes não permite a programação de computações paralelas, pois de seus programas origina um único processo durante a sua execução. Tais linguagens são denominadas seqüenciais. Linguagens que permitem a construção de programas que originam vários processos para serem executados em paralelo são denominadas **linguagens de programação concorrente**. Exemplos deste tipo de linguagem são: Pascal Concorrente, Modula 2, Ada e outras.

A programação concorrente, além de ser intelectualmente atraente e ser essencial ao projeto de sistemas operacionais, também tem aplicações na construção de diversos outros tipos de sistema importantes. Qualquer sistema que deva atender a requisições de serviço que possam ocorrer de forma imprevisível pode ser organizado, convenientemente, para permitir que cada tipo de serviço seja realizado por um dos processos do sistema. Dessa maneira, diversos serviços poderão ser executa-

dos simultaneamente e a utilização dos recursos computacionais será, certamente, mais econômica e eficiente. Exemplos de aplicações deste tipo são sistemas para controle *on-line* de informações (contas bancárias, estoques, etc) e controle de processos externos (processos industriais, processos químicos, rotas de foguetes, etc).

Os processos durante suas execuções requerem operações de E/S que são executadas em dispositivos muito lentos que a UCP, pois os dispositivos periféricos possuem componentes mecânicos, que funcionam a velocidades muito inferiores à dos dispositivos eletrônicos que funcionam à velocidade da luz.

Durante o tempo em que um processo deve ficar esperando a realização de uma operação de E/S, a UCP pode ser entregue a outro processo. Dessa forma, a utilização dos recursos será mais completa e, portanto, mais econômica e mais eficiente. Se um processo passa a maior parte do tempo esperando por dispositivos de E/S, diz-se que o processo é **limitado por E/S** (*I/O-bound*). Se, ao contrário, o processo gasta a maior parte do seu tempo usando a UCP ele é dito **limitado por computação** (*compute-bound* ou *UCP-bound*). Obviamente, processos *I/O-bound* devem ter prioridade sobre processos *UCP-bound*.

Além de uma melhor utilização dos recursos, a multiprogramação permite que as requisições de serviço dos usuários sejam atendidas com menores tempos de resposta. Por exemplo, na situação de um *job* pequeno e prioritário ser submetido após um *job* demorado já ter iniciado a execução, a multiprogramação fará com que o *job* pequeno seja executado em paralelo e termine muito antes do término do *job* longo.

Os sistemas operacionais acionam os dispositivos de E/S através de instruções do tipo *Start I/O* (Iniciar E/S). Se o dispositivo é uma unidade de disco, por exemplo, a instrução faz com que um bloco de setores do disco seja lido para a memória principal. Quando o dispositivo termina a operação, ele manda um sinal de interrupção para a UCP, indicando que está livre para realizar outra operação. Este sinal faz com que o controle da execução vá para o sistema operacional, o qual pode acionar o dispositivo para executar outra operação, antes de devolver a UCP para um processo de usuário.

Durante suas execuções os processos dos usuários, ocasionalmente, através de *traps*, fazem requisições ao sistema operacional (para gravar um setor de disco, por exemplo). Recebendo a requisição, o sistema operacional bloqueia o processo (deixa de dar tempo de UCP a ele) até que a operação requerida seja completada. Quando isto acontece o processo é desbloqueado e volta a competir pela UCP com os demais processos.

Quando um processo está realmente usando a UCP, diz-se que o mesmo está no estado **executando**. Quando está esperando pelo término de um serviço que requereu, diz-se que está no estado **bloqueado**. Quando o processo tem todas as condições para ser executado e só não está em execução porque a UCP está alocada para outro processo, diz-se que o mesmo está no estado **pronto**. O sistema operacional mantém uma lista (fila) dos processos que estão prontos, a chamada **lista de processos prontos**. O diagrama da figura 4.1 mostra como os estados de um processo podem mudar durante a execução.

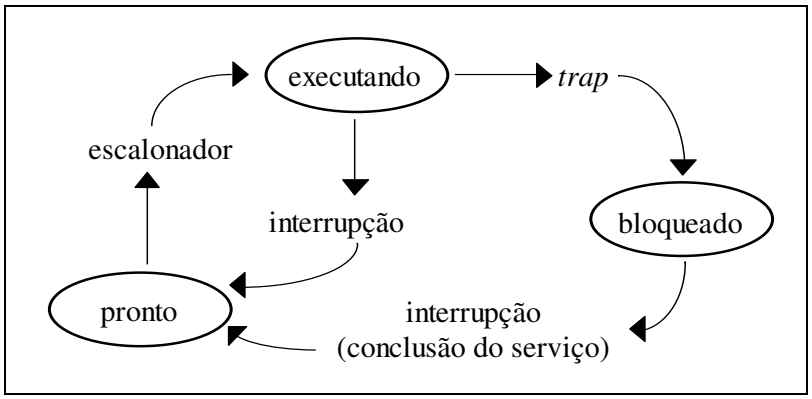


Figura 4.1 - Estados sucessivos de um processo no sistema

O componente do sistema operacional que, após o atendimento de uma interrupção ou *trap*, escolhe o próximo processo a ser executado é denominado **escalonador** de processos (*scheduler*) ou **despachador** de processos (*dispatcher*).

Em geral, um *trap* faz com que o processo fique bloqueado. Entretanto, em algumas ocasiões especiais, quando o sistema operacional pode atender imediatamente a requisição de serviço, o processo pode ser novamente despachado, não ocorrendo o bloqueio.

Quando um *job* é admitido no sistema, um processo correspondente é criado e normalmente inserido no final da *ready list*. O processo se move gradualmente para a cabeça da *ready list*, conforme os processos anteriores a ele forem sendo usados pela UCP.

Quando o processo alcança a cabeça da lista, e quando a UCP torna-se disponível, o processo é dado à UCP e diz-se que foi feito uma transição do estado *ready* para o estado *running*. A transferência da UCP para o primeiro processo da *ready list* é chamada *dispatching*, e é executada pelo *dispatcher* (ou escalonador). Este transição de estado pode ser ilustrada da seguinte forma:

Dispatch(processname): *ready* ➔ *running*

Para prevenir que um processo monopolize o sistema acidentalmente ou propositadamente, o S.O. (Sistema Operacional) tem um relógio interno (*interrupting clock* ou *interval timer*) que faz com que o processo execute somente por um intervalo de tempo específico ou *quantum*. Se o processo voluntariamente não libera a UCP antes de expirar seu intervalo de tempo, o *interrupting clock* gera uma interrupção, dando ao S.O. o controle novamente. O S.O. torna o processo corrente (*running*) em pronto (*ready*) e torna o primeiro processo da *ready list* em corrente. Estas transições de estado são indicadas como:

TimerRunOut(processname): *running* ➔ *ready*
Dispatch(processname): *ready* ➔ *running*

Se um processo corrente iniciar uma operação de I/O antes de expirar o seu *quantum*, o processo corrente voluntariamente libera a UCP (isto é, ele se bloqueia, ficando pendente até completar a operação de I/O). Esta transição de estado é:

Block(processname): *running* ➔ *blocked*

Quando é terminada a operação que fez com que o estado fique bloqueado, este passa para o estado pronto. A transição que faz tal operação é definida como:

WakeUp(processname): *blocked* ➔ *ready*

Deste modo podemos definir quatro possíveis estados de transição:

Dispatch(processname): ready ➔ running
TimerRunOut(processname): running ➔ ready
Block(processname): running ➔ blocked
WakeUp(processname): blocked ➔ ready

Note que somente um estado de transição é inicializado pelo próprio processo — a transição Block os outros três estados de transição são inicializados por entidades externas ao processo.

4.1 O Núcleo do Sistema Operacional

Todas as operações envolvendo processos são controladas por uma porção do sistema operacional chamada de **núcleo**, **core**, ou **kernel**. O núcleo normalmente representa somente uma pequena porção do código que em geral é tratado como sendo todo o sistema operacional, mas é a parte de código mais intensivamente utilizada. Por essa razão, o núcleo ordinariamente reside em armazenamento primário (memória RAM) enquanto outras porções do sistema operacional são chamadas da memória secundária quando necessário.

Uma das funções mais importantes incluídas no núcleo é o processamento de interrupções. Em grandes sistemas multiusuário, uma constante rajada de interrupções é direcionada ao processador. Respostas rápidas a essas interrupções são essenciais para manter os recursos do sistema bem utilizados, e para prover tempos de resposta aceitáveis pelos usuários.

O núcleo desabilita interrupções enquanto ele responde a uma interrupção; interrupções são novamente habilitadas após o processamento de uma interrupção estar completo. Com um fluxo permanente de interrupções, é possível que o núcleo mantenha interrupções desabilitadas por um grande porção de tempo; isto pode resultar em respostas insatisfatórias para interrupções. Entretanto, núcleos são projetados para fazer o “mínimo” processamento possível para cada interrupção, e então passar o restante do processamento de uma interrupção para um processo apropriado do sistema que pode terminar de tratá-las enquanto o núcleo continua apto a receber novas interrupções. Isto significa que as interrupções podem ficar habilitadas durante uma porcentagem muito maior do tempo, e o sistema torna-se mais eficiente em responder a requisições das aplicações dos usuários.

4.1.1 Um Resumo das Funções do Núcleo

Um sistema operacional normalmente possui código para executar as seguinte funções:

Manipulação de interrupções;
Criação e destruição de processos;
Troca de contexto de processos;

- Desacatamento de processos;
- Suspensão e reanimação de processos;
- Sincronização de processos;
- Intercomunicação entre processos;
- Manipulação de PCBs;
- Suporte a atividades de E/S;
- Suporte à alocação e desalocação de armazenamento;
- Suporte ao sistema de arquivos;
- Suporte a um mecanismo de chamada/retorno de procedimentos;
- Suporte a certas funções do sistema de contabilização.

4.2 Escalonamento de Processos

Até agora vimos situações onde tínhamos dois ou mais processos que poderiam estar executando a qualquer momento. Estes processos poderiam estar executando, bloqueados, ou prontos para serem executados. Uma situação adicional, que vimos mais tarde, foi o estado suspenso.

Quando um ou mais processos estão **prontos** para serem executados, o sistema operacional deve decidir qual deles vai ser executado primeiro. A parte do sistema operacional responsável por essa decisão é chamada **escalonador**, e o algoritmo usado para tal é chamado de **algoritmo de escalonamento**. Os algoritmos de escalonamento dos primeiros sistemas, baseados em cartões perfurados e unidades de fita, era simples: ele simplesmente deveria executar o próximo *job* na fita ou leitora de cartões. Em sistemas multi-usuário e de tempo compartilhado, muitas vezes combinados com *jobs batch* em *background*, o algoritmo de escalonamento é mais complexo.

Antes de vermos os algoritmos de escalonamento, vejamos os critérios com os quais eles devem se preocupar:

Justiça: fazer com que cada processo ganhe seu tempo justo de CPU;
Eficiência: manter a CPU ocupada 100% do tempo (se houver demanda);
Tempo de Reposta: minimizar o tempo de resposta para os usuários interativos;
Tempo de *Turnaround*: minimizar o tempo que usuários *batch* devem esperar pelo resultado;

Throughput: maximizar o número de *jobs* processados por unidade de tempo.

Um pouco de análise mostrará que alguns desses objetivos são contraditórios. Para minimizar o tempo de resposta para usuários interativos, o escalonador não deveria

rodar nenhum *job batch* (exceto entre 3 e 6 da manhã, quando os usuários interativos estão dormindo). Usuários *batch* não gostarão deste algoritmo, porque ele viola a regra 4.

Uma complicação que os escalonadores devem levar em consideração é que cada processo é único e imprevisível. Alguns passam a maior parte do tempo esperando por E/S de arquivos, enquanto outros utilizam a CPU por horas se tiverem chance. Quando o escalonador inicia a execução de um processo, ele nunca sabe com certeza quanto tempo vai demorar até que o processo bloqueie, seja por E/S, seja em um semáforo, seja por outro motivo. Para que um processo não execute tempo demais, praticamente todos os computadores possuem um mecanismo de relógio (**clock**) que causa uma interrupção periodicamente. Freqüências de 50 ou 60 Hz são comuns, mas muitas máquinas permitem que o SO especifique esta freqüência. A cada interrupção de relógio, o sistema operacional assume o controle e decide se o processo pode continuar executando ou se já ganhou tempo de CPU suficiente. Neste último caso, o processo é suspenso e a CPU é dada a outro processo.

A estratégia de permitir ao SO temporariamente suspender a execução de processos que estejam querendo executar é chamada de **escalonamento preemptivo**, em contraste com o método **execute até o fim** dos antigos sistemas *batch*. Como vimos até agora, em sistemas preemptivos um processo pode perder a CPU a qualquer momento para outro processo, sem qualquer aviso. Isto gera condições de corrida e a necessidade de semáforos, contadores de eventos, monitores, ou algum outro método de comunicação inter-processos. Por outro lado, uma política de deixar um processo rodar enquanto desejar pode fazer com que um processo que demore uma semana para executar deixe o computador ocupado para os outros usuários durante este tempo.

4.2.1 Escalonamento FCFS ou FIFO

Talvez a disciplina de escalonamento mais simples que exista seja a *First-In-First-Out* - FIFO (o primeiro a entrar é o primeiro a sair). Vários autores referem-se a este algoritmo como FCFS - *First-Come-First-Served* (o primeiro a chegar é o primeiro a ser servido). Processos são despachados de acordo com sua ordem de chegada na fila de processos prontos do sistema. Uma vez que um processo ganhe a CPU, ele roda até terminar. FIFO é uma disciplina não preemptiva. Ela é justa no sentido de que todos os *jobs* são executados, e na ordem de chegada, mas é injusta no sentido que grandes *jobs* podem fazer pequenos *jobs* esperarem, e *jobs* sem grande importância fazem *jobs* importantes esperar. FIFO oferece uma menor variância nos tempos de resposta e é portanto mais previsível do que outros esquemas. Ele não é útil no escalonamento de usuários interativos porque não pode garantir bons tempos de resposta. Sua natureza é essencialmente a de um sistema *batch*.

4.2.2 Escalonamento *Round Robin* (RR)

Um dos mais antigos, simples, justos, e mais largamente utilizados dos algoritmos de escalonamento é o **round robin**. Cada processo recebe um intervalo de tempo, chamado **quantum**, durante o qual ele pode executar. Se o processo ainda estiver executando ao final do *quantum*, a CPU é dada a outro processo. Se um processo

bloqueou ou terminou antes do final do *quantum*, a troca de CPU para outro processo é obviamente feita assim que o processo bloqueia ou termina. *Round Robin* é fácil de implementar. Tudo que o escalonador tem a fazer é manter uma lista de processos *runnable* (que desejam executar), conforme a figura 2.2(a). Quando o *quantum* de um processo acaba, ele é colocado no final da lista, conforme a figura 4.2(b).

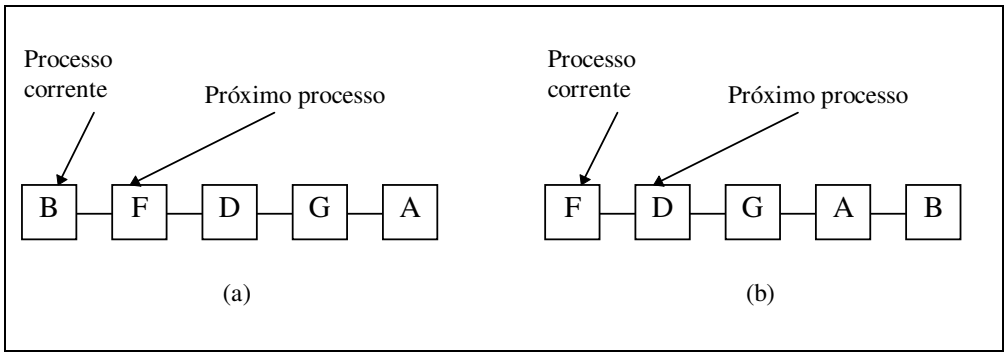


Figura 4.2 - Escalonamento Round Robin. (a) Lista de processos a executar, (b) Lista de processos a executar depois de terminado o *quantum* de 'B'

Assim, o algoritmo *round robin* é semelhante ao FIFO, mas com a diferença de que é preemptivo: os processos não executam até o seu final, mas sim durante um certo tempo, um por vez. Executando sucessivamente em intervalos de tempo o *job* acaba por terminar sua execução em algum momento.

O único aspecto interessante sobre o algoritmo *round robin* é a duração do *quantum*. Mudar de um processo para outro requer um certo tempo para a administração salvar e carregar registradores e mapas de memória, atualizar tabelas e listas do SO, etc. Suponha esta **troca de processos** ou **troca de contexto**, como às vezes é chamada, dure 5 ms. Suponha também que o *quantum* está ajustado em 20 ms. Com esses parâmetros, após fazer 20 ms de trabalho útil, a CPU terá que gastar 5 ms com troca de contexto. Assim, 20% do tempo de CPU é gasto com o *overhead* administrativo.

Para melhorar a eficiência da CPU, poderíamos ajustar o *quantum* para digamos, 500 ms. Agora o tempo gasto com troca de contexto é menos do que 1 %. Mas considere o que aconteceria se dez usuários apertassem a tecla <ENTER> exatamente ao mesmo tempo, disparando cada um processo. Dez processos serão colocados na lista de processo aptos a executar. Se a CPU estiver ociosa, o primeiro começará imediatamente, o segundo não começará antes de ½ segundo depois, e assim por diante. O azarado do último processo somente começará a executar 5 segundos depois do usuário ter apertado <ENTER>, isto se todos os outros processos tiverem utilizado todo o seu *quantum*. Muitos usuários vão achar que o tempo de resposta de 5 segundos para um comando simples é "muita" coisa.
Conclusão: ajustar um *quantum* muito pequeno causa muitas trocas de contexto e diminui a eficiência da CPU, mas ajustá-lo para um valor muito alto causa um tempo de resposta inaceitável para pequenas tarefas interativas. Um *quantum* em torno de 100 ms freqüentemente é um valor razoável.

4.2.3 Escalonamento com Prioridades

O algoritmo *round robin* assume que todos os processos são igualmente importantes. Freqüentemente, as pessoas que possuem e operam centros de computação possuem um pensamento diferente sobre este assunto. Em uma Universidade, por exemplo, as prioridades de processamento normalmente são para a administração em primeiro lugar, seguida de professores, secretárias e finalmente estudantes. A necessidade de se levar em conta fatores externos nos leva ao **escalonamento com prioridades**. A idéia básica é direta: cada processo possui uma prioridade associada, e o processo pronto para executar com a maior prioridade é quem ganha o processador.

Para evitar que processos com alta prioridade executem indefinidamente, o escalonador pode decrementar a prioridade do processo atualmente executando a cada *tick* de relógio (isto é, a cada interrupção de relógio). Se esta ação fizer com que a prioridade do processo se torne menor do que a prioridade do processo que possuía a segunda mais alta prioridade, então uma troca de processos ocorre.

Prioridades podem ser associadas a processos estaticamente ou dinamicamente. Em um computador militar, por exemplo, processos iniciados por generais deveriam começar com a prioridade 100, processos de coronéis com 90, de majores com 80, de capitães com 70, de tenentes com 60, e assim por diante. Alternaticamente, em um centro de computação comercial (incomum hoje em dia), *jobs* de alta prioridade poderiam custar 100 dólares por hora, os de média prioridade a 75 por hora, e os de baixa prioridade a 50 por hora. O sistema operacional UNIX possui um comando, **nice**, que permite a um usuário voluntariamente reduzir a prioridade de um processo seu, de modo a ser gentil (*nice*) com os outros usuários. Na prática, ninguém utiliza este comando, pois ele somente permite baixar a prioridade do processo. Entretanto, o superusuário UNIX pode aumentar a prioridade de processos.

Prioridades podem também ser atribuídas dinamicamente pelo sistema para atingir certos objetivos do sistema. Por exemplo, alguns processos são altamente limitados por E/S, e passam a maior parte do tempo esperando por operações de E/S. Sempre que um desses processos quiser a CPU, ele deve obtê-la imediatamente, para que possa iniciar sua próxima requisição de E/S, e deixá-la sendo feita em paralelo com outro processo realmente processando. Fazer com que processos limitados por E/S esperem um bom tempo pela CPU significa deixá-los um tempo demasiado ocupando memória. Um algoritmo simples para prover um bom serviço a um processo limitado por E/S é ajustar a sua prioridade para $1/f$, onde f é a fração do último *quantum* de processador que o processo utilizou. Um processo que utilizou somente 2 ms do seu *quantum* de 100 ms ganharia uma prioridade 50, enquanto um processo que executou durante 50 ms antes de bloquear ganharia prioridade 2, e um processo que utilizou todo o *quantum* ganharia uma prioridade 1.

É freqüentemente conveniente agrupar processos em classes de prioridade e utilizar escalonamento com prioridades entre as classes, mas *round robin* dentro de cada classe. Por exemplo, em um sistema com quatro classes de prioridade, o escalonador executa os processos na classe 4 segundo a política *round robin* até que não haja mais processos na classe 4. Então ele passa a executar os processos de classe

3 também segundo a política *round robin*, enquanto houverem processos nesta classe. Então executa processos da classe 2 e assim por diante. Se as prioridades não forem ajustadas de tempos em tempos, os processos nas classes de prioridades mais baixas podem sofrer o fenômeno que chamamos **starvation** (o processo nunca recebe o processador, pois sua vez nunca chega).

4.2.4 Multilevel Feedback Queues

Quanto um processo ganha a CPU, especialmente quando ele ainda não pôde estabelecer um padrão de comportamento, o escalonador não tem idéia do quantidade precisa de tempo de CPU que o processo precisará. Processos limitados por E/S geralmente usam a CPU brevemente antes de gerar em pedido de E/S. Processos limitados por CPU poderiam utilizar a CPU por horas se ela estivesse disponível para eles em um ambiente não preemptivo.

Um mecanismo de escalonamento deveria:
favorecer pequenos *jobs*;
favorecer *jobs* limitados por E/S para atingir uma boa utilização dos dispositivos de E/S; e
determinar a natureza de um *job* tão rápido quanto possível e escalonar o *job* de acordo.

Multilevel feedback queues (filas multi-nível com retorno) fornecem uma estrutura que atinge esses objetivos. O esquema é ilustrado na figura 4.3:

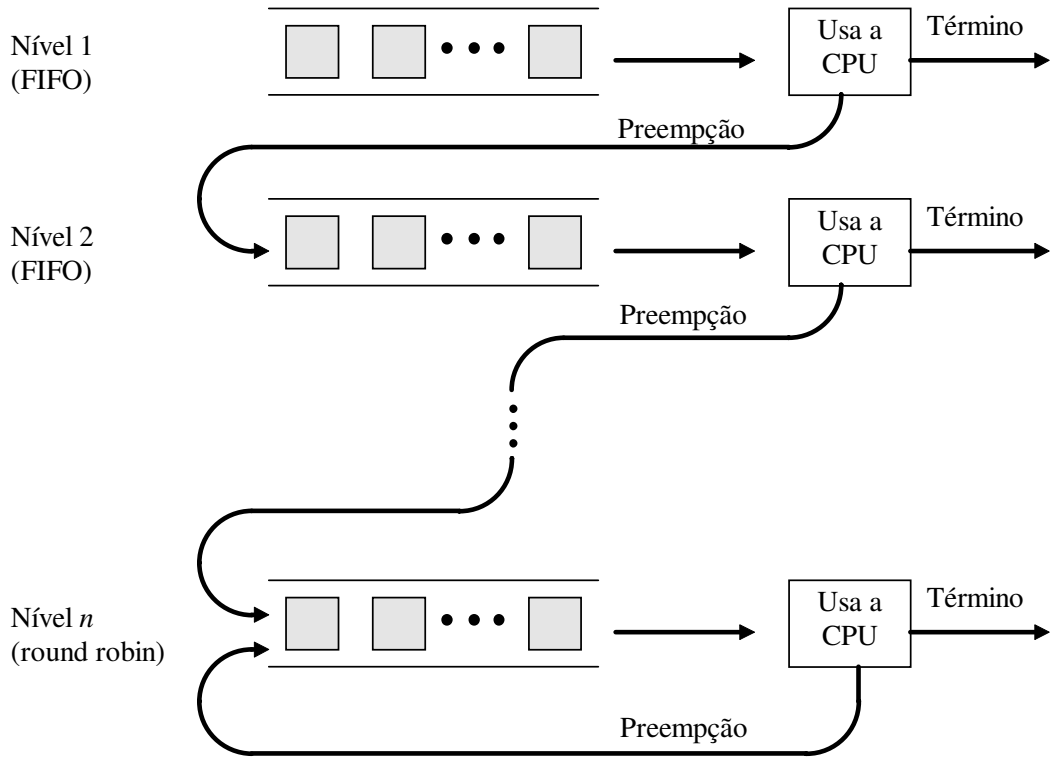


Figura 4.3 - Filas Multinível com Retorno

Um novo processo entra na rede de filas ao final da fila do topo. Ele se move através desta fila segundo uma política FIFO até que ganhe a CPU. Se o *job* termina ou desiste da CPU para esperar um término de E/S ou outro evento, ele deixa a rede de filas. Se o *quantum* expira antes do processo voluntariamente desistir da CPU, o processo é colocado de volta no final da fila um nível abaixo. O processo avança nesta fila, e em algum momento atinge a cabeça da fila. No momento em que não houverem processos na primeira fila, ele ganha a CPU novamente. Se ele ainda utiliza todo o *quantum*, ele vai descendo para as filas de níveis inferiores. Normalmente, a fila de nível mais baixo possui uma política *round robin* para que todos os processos terminem de executar de uma maneira ou outra.

Em muitos sistemas de filas multi-nível, o *quantum* dado ao processo conforme ele se move para as filas de níveis inferiores é aumentado. Assim, quanto mais um processo permanece no sistema de filas, maior o seu *quantum*. Entretanto ele passa a não ganhar a CPU com tanta freqüência, porque as filas superiores possuem prioridade maior. Um processo em uma dada fila não pode executar até que as filas superiores estejam vazias. Um processo em execução é suspenso em favor de um processo que chegue em uma fila superior.

Considere como tal mecanismo responde a diferentes tipos de processos. O mecanismo deveria favorecer processos limitados por E/S para atingir boa utilização dos dispositivos e bons tempos de resposta aos usuários interativos. Realmente isso funciona porque um processo limitado por E/S vai entrar na primeira fila e rapidamente ganhar a CPU. O *quantum* da primeira fila é ajustado para que a maioria dos *jobs* limitados por E/S tenham tempo de fazer sua requisição de E/S. Quando o processo faz a requisição de E/S, ele deixa a rede de filas, tendo recebido o tratamento desejado.

Agora considere um processo limitado por CPU que necessita de um grande tempo de CPU. Ele entra a rede de filas no nível mais alto, recebendo rapidamente seu primeiro *quantum* de CPU, mas quando ele expira, o processo é movido para a fila inferior. Agora o processo tem prioridade inferior aos da fila superior, mas eventualmente ele recebe a CPU, ganhando um *quantum* maior do que o anterior. Conforme o processo ainda precise de CPU, ele vai caminhando pelas filas, até chegar à fila de mais baixo nível, onde ele circula por uma fila *round robin* até que tenha terminado.

Filas Multi-nível com retorno são ideais para separar processos em categorias baseadas na sua necessidade por CPU. Em um sistema de tempo compartilhado, cada vez que o processo deixe a rede de filas, ele pode ser marcado com a identificação do nível da fila onde ele esteve pela última vez. Quando o processo reentra no sistema de filas, ele pode ser enviado diretamente para a fila onde ele anteriormente completou sua execução, de forma que um processo retornando para as filas não interfira no desempenho dos processos das filas de níveis mais altos.

Se processos são sempre colocados de volta na rede de filas no nível mais alto que eles ocuparam da última vez que estiveram no sistema de filas, será impossível para o sistema responder a mudanças no processo, como por exemplo, deixando de ser limitado por CPU para ser limitado por E/S. Este problema pode ser resolvido mar-

cando também o processo com o seu tempo de permanência na rede de filas na última vez em que lá esteve. Assim, quando o processo reentra no sistema de filas, ele pode ser colocado no lugar correto. Dessa forma, se o processo está entrando em uma nova fase na qual ela deixa de ser limitado por CPU para ser limitado por E/S, inicialmente ele vai sofrer uma penalidade pelo sistema, mas da próxima vez o algoritmo perceberá a mudança de comportamento do processo. Uma outra maneira de responder a mudanças no comportamento de um processo é colocá-lo em um nível de filas cima do qual esteve se ele voluntariamente desistir da CPU antes do término do seu *quantum*.

O mecanismo de filas multi-nível com retorno é um bom exemplo de um **mecanismo adaptativo**, que responde a mudanças de comportamento do sistema que ele controla. Mecanismos adaptativos normalmente requerem um maior *overhead* do que os não adaptativos, mas a sensibilidade a mudanças torna o sistema mais ágil e justifica o *overhead* adicional.

Uma variação comum deste algoritmo é ter os processos circulando em várias filas *round robin*. O processo circula pela primeira fila um certo número de vezes, depois desce um nível, circulando um número maior de vezes, e assim por diante.

4.2.5 Escalonamento com Prazos

No **escalonamento com prazos** certos *jobs* são escalonados para serem completados até uma certa data ou hora, ou um prazo. Esses *jobs* podem ter alta importância se entregues tem tempo, ou podem não ter utilidade alguma se terminarem de ser processados além do tempo previsto no prazo. O usuário normalmente paga um "prêmio" para ter seu *job* completado em tempo. Este tipo de escalonamento é complexo por muitas razões:

usuário deve fornecer os requerimentos precisos de recursos do *job* previamente. Tal informação raramente está disponível;

- sistema deve rodar o *job* com prazo sem degradar o serviço para os outros usuários;
- sistema deve cuidadosamente planejar seus requerimentos de recursos durante o prazo. Isto pode ser difícil porque novos *jobs* podem chegar e adicionar uma demanda imprevisível no sistema;
- Se muitos *jobs* com prazos são supostos existirem ao mesmo tempo, o escalonamento com prazos poderia se tornar tão complexo que métodos sofisticados de otimização poderiam ser necessários para garantir que as metas sejam atingidas;
- intensivo gerenciamento de recursos requerido pelo escalonamento com prazos pode gerar um *overhead* substancial. Mesmo que os usuários dos *jobs* com prazos estejam aptos a pagar uma taxa suficientemente alta pelos serviços recebidos, o consumo total de recursos do sistema pode se tornar tão alto que o resto da comunidade de usuários poderia ter um serviço degradado. Tais conflitos devem ser considerados cuidadosamente por projetistas de sistemas operacionais.

4.2.6 Escalonamento *Shortest-Job-First* (SJF)

Shortest-job-first (o menor *job* primeiro) é um algoritmo não preemptivo no qual o *job* na fila de espera com o menor tempo total estimado de processamento é executado em seguida. SJF reduz o tempo médio de espera sobre o algoritmo FIFO. Entretanto, os tempos de espera tem uma variância muito grande (são mais imprevisíveis) do que no algoritmo FIFO, especialmente para grandes *jobs*.

SJF favorece *jobs* pequenos em prejuízo dos *jobs* maiores. Muitos projetistas acreditam que quanto mais curto o *job*, melhor serviço ele deveria receber. Não há um consenso universal quanto a isso, especialmente quando prioridades de *jobs* devem ser consideradas.

SJF seleciona *jobs* para serviço de uma maneira que garante que o próximo *job* irá completar e deixar o sistema o mais cedo possível. Isto tende a reduzir o número de *jobs* esperando, e também reduz o número de *jobs* esperando atrás de grandes *jobs*. Como resultado, SJF pode minimizar o tempo médio de espera conforme eles passam pelo sistema.

O problema óbvio com SJF é que ele requer um conhecimento preciso de quanto tempo um *job* demorará para executar, e esta informação não está usualmente disponível. O melhor que SJF pode fazer é se basear na estimativa do usuário de tempo de execução. Em ambientes de produção onde os mesmos *jobs* rodam regularmente, pode ser possível prover estimativas razoáveis. Mas em ambientes de desenvolvimento, os usuários raramente sabem durante quanto tempo seus programas vão executar.

Basear-se nas estimativas dos usuários possui uma ramificação interessante. Se os usuários sabem que o sistema está projetado para favorecer *jobs* com tempos estimados de execução pequenos, eles podem fornecer estimativas com valores menores que os reais. O escalonador pode ser projetado, entretanto, para remover esta tentação. O usuário pode ser avisado previamente que se o *job* executar por um tempo maior do que o estimado, ele será abortado e o usuário terá que ser cobrado pelo trabalho. Uma segunda opção é rodar o *job* pelo tempo estimado mais uma pequena percentagem extra, e então salvá-lo no seu estado corrente de forma que possa ser continuado mais tarde. O usuário, é claro, teria que pagar por este serviço, e ainda sofreria um atraso na completude de seu *job*. Outra solução é rodar o *job* durante o tempo estimado a taxas de serviços normais, e então cobrar uma taxa diferenciada (mais cara) durante o tempo que executar além do previsto. Dessa forma, o usuário que fornecer tempos de execução sub-estimados pode pagar um preço alto por isso.

SJF, assim como FIFO, é não preemptivo e, portanto não é útil para sistemas de tempo compartilhado nos quais tempos razoáveis de resposta devem ser garantidos.

4.3 Comunicação Entre Processos (IPC)

Processos são concorrentes se eles existem no mesmo tempo. Processos concorrentes podem funcionar completamente independentes um dos outros, ou eles po-

dem ser assíncronos, o que significa que eles ocasionalmente necessitam de sincronização e cooperação.

Veremos vários problemas e também soluções para o assincronismo de processos concorrentes.

Processos concorrentes implicam em compartilhamento de recursos do sistema, tais como arquivos, registros, dispositivos de I/O e áreas de memória. Um sistema operacional multiprogramável deve fornecer mecanismos de **sincronização de processos**, para garantir a **comunicação inter-processos** (IPC - *Inter-Process Communication*) e o acesso a recursos compartilhados de forma organizada.

4.3.1 Processamento Paralelo

Conforme o *hardware* continua a cair em preços e tamanho, cada vez tornam-se mais evidentes as tendências para multiprocessamento, processamento distribuído e máquinas massivamente paralelas. Se certas operações podem ser logicamente executadas em paralelo, os computadores paralelos irão fisicamente executá-las em paralelo, mesmo que o **nível de paralelismo** seja da ordem de milhares ou talvez milhões de atividades concorrentes. Isto pode resultar em um aumento significativo de performance sobre os computadores seqüenciais (com um único processador).
O processamento paralelo é um assunto interessante e ao mesmo tempo complexo por várias razões. As pessoas parecem melhor focalizar sua atenção para uma atividade por vez do que pensar em paralelo. Um exemplo disso é tentar fazer uma pessoa ler dois livros ao mesmo tempo: uma linha de um, uma linha do outro, e assim por diante.

É difícil e denota tempo determinar quais atividades podem e quais não podem ser executadas em paralelo. Programas paralelos são muito mais difíceis de *debugar* do que programas seqüenciais depois de supostamente consertar um *bug*, é praticamente impossível reproduzir a seqüência exata de eventos que fez com que o erro aparecesse, dessa forma então impossibilitando certificar, com certeza, de que o erro foi corretamente removido.

Processos assíncronos podem ocasionalmente interagir um com o outro, e essas interações podem ser complexas.

Finalmente, é muito mais difícil provas que programas paralelos estão corretos, do que programas seqüenciais. A prova da corretude de programas envolve geralmente testes exaustivos, e as possibilidades de execução em programas paralelos podem ser infinitas, sendo impossível a exaustão de todas as possibilidades.

4.3.1.1 Comandos PARBEGIN e PAREND (Dijkstra)

Várias linguagens de programação paralela têm aparecido na literatura. Estas geralmente envolvem um par de comandos, da seguinte forma:

Um comando indicando que a execução seqüencial passa a ser dividida em várias seqüências de execução em paralelo (linhas de controle ou execução); e um co-

mando indicando que certas seqüências de execução paralelas devem se juntar para a execução seqüencial continuar.

Estes comandos ocorrem sempre aos pares e são comumente chamados **parbegin**/**parend** (início e fim de execução paralela), ou **cobegin**/**coend** (início e fim de execução concorrente). Utilizaremos parbegin/parend, conforme sugerido por Dijkstra (1965), na seguinte forma geral:

```
parbegin
        comando-1;
        comando-2;
          .
          .
          .
        comando-n
parend
```

Suponhamos que um programa em execução encontre a construção *parbegin*. Isto vai causar uma divisão de controle da execução entre os vários comandos seguintes, possivelmente entre vários processadores (se a máquina for multiprocessada). Quando o comando *parend* é atingido, a execução só prossegue até que todos os sub-comandos concorrentes tenham terminado. A partir deste ponto, a execução passa novamente a ser seqüencial.

Considere o seguinte exemplo, onde o operador ** é o operador de exponenciação:
x := (-b + (b ** 2 - 4 * a * c) ** .5) / (2 * a)

Em uma máquina com processamento seqüencial, a expressão acima seria desmembrada em várias ações:

```
1       b ** 2
2       4 * a
3       (4 * a) * c
4       (b ** 2) - (4 * a * c)
5       (b ** 2 - 4 * a *c) ** .5
6       -b
7       (-b) + ((b ** 2 - 4 * a * c) ** .5)
8       2 * a
9       (-b + (b ** 2 - 4 * a * c) ** .5) / (2 * a)
```

Em um sistema que suporte processamento paralelo, a mesma expressão deveria ser executada da seguinte maneira:

```
parbegin
 temp1 := -b;
 temp2 := b ** 2;
 temp3 := 4 * a;
 temp4 := 2 * a
parend;
```

```
temp5 := temp3 * c;
temp5 := temp2 - temp5;
temp5 := temp5 ** .5;
temp5 := temp1 + temp5;
x := temp5 / temp4
```

Neste caso, as operações entre o *parbegin* e o *parend* são executadas em paralelo as operações restantes devem ser executadas seqüencialmente. Pela execução das instruções em paralelo, é possível reduzir substancialmente o tempo real de execução do programa.

4.3.1.2 Comandos FORK e JOIN (Conway e Dennis)

Utilizados pelo sistema operacional UNIX, e também implementados em algumas linguagens de programação, como PL/1.

```
program A;                      program B;
        ...                             ...
        fork B;                 ...
        ...                             ...
        join B;         end.
        ...
end.
```

O programa A começa a ser processado e, ao encontrar o comando FORK, faz com que seja criado um outro processo (ou subprocesso) para a execução do programa B, concorrentemente a A. O comando JOIN permite que o programa A sincronize-se com B, ou seja, quando o programa A encontrar o comando JOIN, só continuará a ser processado após o término de B.

4.3.2 Exclusão Mútua

Considere um sistema com muitos terminais em tempo compartilhado. Assuma que os usuários terminam cada linha de texto que eles digitam com a tecla <ENTER>. Suponha que seja desejável monitorar continuamente o número total de linhas que os usuários entraram no sistema desde o início do dia. Assuma que cada terminal de usuário seja monitorado por um processo diferente. Toda vez que um destes processos recebe uma linha do terminal, ele incrementa de 1 uma variável global compartilhada do sistema, chamada LINHAS. Considere o que aconteceria se dois processos tentassem incrementar LINHAS simultaneamente. Assuma que cada processo possui sua própria cópia do seguinte código:

```
LOAD    LINHAS      ; Lê a variável linhas no registrador acumulador
ADD     1           ; Incrementa o registrador acumulador
STORE   LINHAS      ; Armazena o conteúdo do acumulador na variável
```

Suponhamos que LINHAS tenha o valor atual 21687. Agora suponhamos que o primeiro processo execute as instruções LOAD e ADD, deixando então o valor 21688 no acumulador. Então o processo perde o processador (após o término de seu quan-

tum) para o segundo processo. O segundo processo então executa as três instruções, fazendo com que a variável linhas tenha o valor 21688. Ele então perde o processador para o primeiro processo que continua executando de onde tinha parado, e, portanto executando a instrução STORE, e armazenando 21688 na variável LINHAS. Devido à falta de comunicação entre os processos, o sistema deixou de contar uma linha o valor correto seria 21689.

O problema está em dois ou mais processos escreverem em uma variável compartilhada. Vários processos poderiam estar lendo uma variável compartilhada sem problemas. Entretanto, quando um processo lê uma variável que outro processo está escrevendo, ou quando um processo tenta escrever em uma variável que outro processo também esteja escrevendo, resultados inesperados podem acontecer.

O problema pode ser resolvido dando a cada processo **acesso exclusivo** à variável LINHAS. Enquanto um processo incrementa a variável, todos os outros processos desejando fazê-lo no mesmo instante deverão esperar; quando o primeiro processo terminar o acesso à variável compartilhada, um dos processos passa a ter acesso à variável. Assim, cada processo acessando a variável exclui todos outros de fazê-lo simultaneamente. Isto é chamado de **exclusão mútua**. Como veremos, os processos em espera deverão ser gerenciados de forma a esperarem um tempo razoavelmente curto.

4.3.3 Regiões Críticas

A exclusão mútua somente precisa estar presente nos instantes em que os processos acessam dados compartilhados modificáveis — quando os processos estão executando operações que não conflitam um com o outro, eles deveriam ser liberados para processarem concorrentemente. Quando um processo está acessando dados compartilhados modificáveis, é dito que o processo está em sua **região crítica** ou **seção crítica**.

Fica claro, que para evitarmos problemas como o mostrando acima, é necessário garantir que quando um processo está em sua região crítica, todos os outros processos (pelo menos aqueles que acessam a mesma variável compartilhada modificável) sejam excluídos de suas respectivas regiões críticas.

Enquanto um processo está em sua região crítica, outros processos podem certamente continuar sua execução fora de suas regiões críticas. Quando um processo deixa sua região crítica, um dos processos esperando para entrar em sua própria região crítica pode prosseguir. Garantir a exclusão mútua é um dos principais problemas em programação concorrente. Muitas soluções foram propostas: algumas baseadas em *software* e outras baseadas em *hardware*; algumas em baixo nível, outras em alto nível; algumas requerendo cooperação voluntária entre processos, outras exigindo uma rígida aderência a protocolos estritos.

Um processo dentro de uma região crítica está em um estado muito especial. O processo possui acesso exclusivo aos dados compartilhados modificáveis, e todos os outros processos desejando acessá-los devem esperar. Assim, regiões críticas devem executar o mais rápido possível, um processo não deve passar para o estado

bloqueado dentro da região crítica, e regiões críticas devem ser cuidadosamente codificadas (para evitar, por exemplo, *loops* infinitos).

4.3.4 Primitivas de Exclusão Mútua

O programa concorrente apresentado a seguir, implementa o mecanismo de contagem de linhas de texto digitadas pelos usuários, descrito anteriormente. A linguagem utilizada é uma pseudo-linguagem com sintaxe semelhante à linguagem Pascal, mas com mecanismos de paralelismo. Para maior simplicidade, vamos assumir que somente dois processos concorrentes nos programas apresentados nesta e nas seções seguintes. Manipular *n* processos concorrentes é consideravelmente mais complexo.

```
program exclusão_mútua;
var linhas_digitadas: integer;
procedure processo_um;
  while true do
    begin
      leia_próxima_linha_do_terminal;
      entermutex;
        linhas_digitadas := linhas_digitadas + 1;
      exitmutex;
      processe_a_linha;
    end;
procedure processo_dois;
  while true do
    begin
      leia_próxima_linha_do_terminal;
      entermutex;
        linhas_digitadas := linhas_digitadas + 1;
      exitmutex;
      processe_a_linha;
    end;
begin
  linhas_digitadas := 0;
  parbegin
    processo_um;
    processo_dois;
  parend;
end.
```

O par de construções **entermutex** / **exitmutex** introduzidos no exemplo delimitam o código da região crítica de cada processo. Essas operações são normalmente chamadas de **primitivas de exclusão mútua**, e cada linguagem que as suporte pode implementá-las à sua maneira.

No caso de dois processos, como no exemplo, as primitivas operam da seguinte maneira. Quando o processo_um executa **entermutex**, se o processo_dois não está em sua região crítica, então o processo_um entra em sua região crítica, acessa a

variável e então executa **exitmutex** para indicar que ele deixou sua região crítica.

Se o processo_dois está em sua região crítica quando o processo_um executa **entermutex**, então o processo_um espera até que o processo_dois execute **exitmutex**. Então o processo_um pode prosseguir e entrar em sua região crítica.

Se tanto o processo_um como o processo_dois executam simultaneamente **entermutex**, então somente é liberado para prosseguir, enquanto o outro permanece esperando. Por enquanto, vamos assumir que o "vencedor" é escolhido aleatoriamente pelo sistema operacional.

4.3.5 Implementação de Primitivas de Exclusão Mútua

A implementação das primitivas **entermutex** (entrada no código de exclusão mútua) e **exitmutex** (saída do código de exclusão mútua) deve satisfazer as seguintes restrições:

A solução deve ser implementada puramente em *software* em uma máquina sem suporte a instruções específicas para exclusão mútua.
Cada instrução da máquina é executada indivisivelmente, isto é, uma vez que uma instrução começa a ser executada, ela termina sem nenhuma interrupção.

Se múltiplos processadores tentam acessar o mesmo item de dados, assumimos que um recurso de *hardware* chamado **storage interlock** (interbloqueio de armazenamento) resolve qualquer conflito. O **storage interlock** seqüencializa as referências conflitantes a um item de dados, isto é, as referências acontecem uma por vez. Assumimos também que as várias referências são servidas em uma ordem aleatória.

Nenhuma suposição deve ser feita sobre as velocidades relativas dos processos concorrentes assíncronos.

Processos operando fora de suas regiões críticas não podem evitar que outros processos entrem em suas próprias regiões críticas.

Processos não devem ter sua vez de entrar em sua região crítica indefinidamente adiada.

Uma implementação elegante em *software* de exclusão mútua foi pela primeira vez apresentada pelo matemático alemão Dekker. Esta solução implementa exclusão mútua com "espera ocupada" (*busy wait*), pois os processos ficam em um *loop* infinito fazendo testes até conseguirem entrar nas suas regiões críticas. O algoritmo de Dekker foi analisado e apresentado por diversos autores da área de sistemas operacionais. Um deles, G. L. Peterson, apresentou mais tarde um algoritmo mais simplificado do algoritmo original de Dekker.

Entretanto, a solução de Dekker somente funciona para exclusão mútua entre dois processos. Outras soluções foram inventadas, e contam com algum suporte do sistema operacional, o que facilita a vida do programador. Vejamos a seguir estas soluções.

4.3.6 Exclusão Mútua para *N* Processos

Em 1965, Dijkstra foi o primeiro a apresentar uma solução em *software* para a implementação de primitivas de exclusão mútua para *n* processos. Knuth respondeu em 1966 com uma solução que eliminou a possibilidade de adiamento indefinido existente no algoritmo de Dijkstra, mas que ainda permitia que um processo pudesse sofrer um longo atraso na hora de entrar em sua região crítica.

Isto desencadeou várias tentativas de encontrar algoritmos com menores atrasos. Eisenberg e McGuire (1972) apresentaram uma solução que garantia que em uma situação com *n* processos, um desses processos conseguiria entrar em sua região crítica com no máximo *n - 1* tentativas.

Em 1974, Lamport desenvolveu uma solução que é particularmente aplicável a sistemas de processamento distribuído. O algoritmo utiliza um sistema do tipo "pegue uma senha", da mesma forma que algumas padarias da época utilizavam, e por isso foi chamado de *Lamport's Bakery Algorithm* (algoritmo de padaria de Lamport).

Brinch Hansen (1978) também discute controle de concorrência entre processos distribuídos. Burns, et alli (1982), oferece uma solução utilizando uma única variável compartilhada. Carvalho e Roucairol (1983) discutem a garantia de exclusão mútua em redes de computadores.

4.3.7 Semáforos

Dijkstra conseguiu abstrair as principais noções de exclusão mútua em seu conceito de **semáforos**. Um semáforo é uma **variável protegida** cujo valor somente pode ser acessado e alterado pelas operações **P** e **V**, e por uma operação de inicialização que chamaremos **inicializa_semáforo**. **Semáforos binários** podem assumir somente os valores 0 ou 1. **Semáforos contadores** (também chamados de **semáforos genéricos**) podem assumir somente valores inteiros não negativos.

A operação **P** no semáforo **S**, denotada por **P(S)**, funciona da seguinte maneira:

```
if S > 0 then
  S := S - 1
else
  (espera no semáforo S)
```

A operação **V** no semáforo **S**, denotada por **V(S)**, funciona da seguinte maneira:

```
if (um ou mais processos estão esperando em S) then
  (deixe um desses processos continuar)
else
  S := S + 1;
```

Nós podemos assumir que existe uma política de enfileiramento **FIFO** (*first-in-first-out* - o primeiro a entrar é o primeiro a sair) para os processos esperando para uma operação **P(S)** completar.

Assim como **testandset**, as operações **P** e **V** são indivisíveis. A exclusão mútua no semáforo S é garantida pelas operações **P(S)** e **V(S)**. Se vários processos tentam executar **P(S)** simultaneamente, somente um deles poderá prosseguir. Os outros ficarão esperando, mas a implementação de **P** e **V** garante que os processos não sofrerão **adiamento indefinido**.

Semáforos e operações com semáforos podem ser implementados tanto em *software* quanto em *hardware*. Eles são comumente implementados no núcleo do sistema operacional, onde a mudança de estado dos processos é controlada. O exemplo a seguir ilustra o uso de semáforos para implementar exclusão mútua. Neste exemplo, P(ativo) é equivalente a **entermutex** e V(ativo) é equivalente a **exitmutex**.

```
program exemplo_semáforo;
var ativo: semaphore;

procedure processo_um;
begin
  while true do
  begin
    algumas_funcoes_um;
    P(ativo);
    regiao_critica_um;
    V(ativo);
    outras_funcoes_um;
  end
end;


procedure processo_dois;
begin
  while true do
  begin
    algumas_funcoes_dois;
    P(ativo);
    regiao_critica_dois;
    V(ativo);
    outras_funcoes_dois;
  end
end;

begin
  inicializa_semaforo(ativo, 1);
  parbegin
    processo_um;
    processo_dois
  parend
end.
```

4.3.7.1 Sincronização de Processos com Semáforos

Quando um processo invoca uma requisição de E/S, ele se auto-bloqueia para esperar o término da operação de E/S. Alguns outros processos devem "acordar" o processo bloqueado. Tal interação é um exemplo de um **protocolo bloqueia/acorda** (*block/wakeup*).

Mais genericamente, suponha que um processo queira ser notificado sobre a ocorrência de um evento particular. Suponha que algum outro processo seja capaz de detectar que esse evento ocorreu. O exemplo seguinte mostra como operações de semáforo podem ser usadas para implementar um mecanismo de sincronização simples *block/wakeup*.

```
program block_wakeup;
var evento_de_interesse: semaphore;

procedure processo_um;
begin
  algumas_funcoes_um;
  P(evento_de_interesse);
  outras_funcoes_um
end;

procedure processo_dois;
begin
  algumas_funcoes_dois;
  V(evento_de_interesse);
  outras_funcoes_dois
end;

begin
  inicializa_semaforo(evento_de_interesse, 0);
  parbegin
    processo_um;
    processo_dois
  parend
end.
```

O processo_um executa algumas_funcoes_um e então executa P(evento_de_interesse) para esperar (*wait*) até que o evento aconteça. O semáforo foi inicializado em zero para que inicialmente o processo espere. Eventualmente o processo_dois executa V(evento_de_interesse) para sinalizar (*signal*) que o evento ocorreu. Isto permite que o processo_um prossiga (com o semáforo ainda em zero).

Note que esse mecanismo funciona mesmo que o processo_dois detecte e sinalize o evento com V(evento_de_interesse) antes que o processo_um execute P(evento_de_interesse) - o semáforo será incrementado de 0 para 1, então P(evento_de_interesse) irá simplesmente decrementar o semáforo de 1 para 0, e o processo_um irá presseguir sem ter que esperar pelo evento.

4.3.7.2 A Relação Produtor-Consumidor

Em um programa seqüencial, quando um procedimento chama um outro e lhe passa dados como parâmetros, os procedimentos são partes de um único processo eles não operam concorrentemente. Mas quando um processo passa dados para outro processo, os problemas são mais complexos. Tal transmissão é um exemplo de **comunicação inter-processos** (*interprocess communication* - IPC).

Considere a seguinte relação produtor-consumidor. Suponha que um processo, um **produtor**, esteja gerando informação que um segundo processo, o **consumidor**, esteja utilizando. Suponha que eles se comunicam por uma única variável inteira compartilhada, chamada numero. O produtor faz alguns cálculos e então escreve o resultado em numero; o consumidor lê o dado de numero e o imprime.

É possível que os processos produtor e consumidor executem sem problemas, ou que suas velocidades sejam incompatíveis. Se cada vez que o produtor depositar um resultado em numero o consumidor puder lê-lo e imprimí-lo, então o resultado impresso certamente representará a seqüência de números gerados pelo produtor.

Mas suponha que as velocidades dos processos sejam incompatíveis. Se o consumidor estiver operando mais rápido que o produtor, o consumidor poderia ler e imprimir o mesmo número duas vezes (ou talvez várias vezes) antes que o produtor depositasse o próximo número. Se o produtor estiver operando mais rápido que o consumidor, o produtor poderia sobrescrever seu resultado prévio sem que o consumidor tivesse a chance de lê-lo e imprimí-lo; um produtor rápido poderia de fato fazer isto várias vezes de forma que vários resultados poderiam ser perdidos.

Obviamente, o comportamento que desejamos aqui é que o produtor e o consumidor operem de tal forma que dados escritos para numero nunca sejam perdidos ou duplicados. A garantia de tal comportamento é um exemplo de **sincronização de processos**.

Abaixo temos um exemplo de um programa concorrente que utiliza operações com semáforos para implementar uma relação produtor-consumidor.

```
program relacao_produtor_consumidor;
var numero: integer;
   numero_depositado: semaphore;
   numero_retirado: semaphore;

procedure processo_produtor;
var proximo_resultado: integer;
begin
  while true do
  begin
    calcule_proximo_resultado;
    P(numero_retirado);
    numero := proximo_resultado;
    V(numero_depositado)
```

```
  end
end;

procedure processo_consumidor;
var proximo_resultado: integer;
begin
  while true do
  begin
    calcule_proximo_resultado;
    P(numero_depositado);
    proximo_resultado := numero;
    V(numero_retirado);
    write(proximo_resultado)
  end
end;

begin
  inicializa_semaforo(numero_depositado, 0);
  inicializa_semaforo(numero_retirado, 1);
  parbegin
    processo_produtor;
    processo_consumidor
  parend
end.
```

Dois semáforos foram utilizados: numero_depositado é sinalizado (operação V) pelo produtor e testado (operação P) pelo consumidor; o consumidor não pode prosseguir até que um número seja depositado em numero. O processo consumidor sinaliza (operação V) numero_retirado e o produtor o testa (operação P); o produtor não pode prosseguir até que um resultado existente em numero seja retirado. Os ajustes iniciais dos semáforos forçam o produtor a depositar um valor em numero antes que o consumidor possa prosseguir. Note que o uso de semáforos neste programa força a sincronização passo a passo; isto é aceitável porque só existe uma única variável compartilhada. É comum em relações produtor-consumidor que se tenha um *buffer* com espaço para várias variáveis. Com tal arranjo, o produtor e o consumidor não precisam executar à mesma velocidade na sincronização passo a passo. Ao contrário, um produtor rápido pode depositar diversos valores enquanto o consumidor está inativo, e um consumidor rápido pode retirar diversos valores enquanto o produtor está inativo. Investigaremos este tipo de relação com mais detalhes no futuro.

4.3.7.3 Semáforos Contadores

Semáforos contadores são particularmente úteis quando um recurso deve ser alocado a partir de um *pool* (agrupamento) de recursos idênticos. O semáforo é inicializado com o número de recursos no agrupamento. Cada operação P decrementa o semáforo de 1, indicando que um recurso foi removido do agrupamento e está em uso por um processo. Cada operação V incrementa o semáforo de 1, indicando que um processo retornou um recurso ao agrupamento, e o recurso pode ser realocado para outro processo. Se uma operação P é executada quando o semáforo possui o valor

zero, então o processo deve esperar até que um recurso seja liberado com uma operação V.

4.3.7.4 Implementando Semáforos, P e V

Dado o algoritmo de Dekker e a disponibilidade de uma instrução de máquina **testandset**, é praticamente direto implementar P e V utilizando **busy waiting**. Mas esta técnica pode não ser boa porque desperdiça recursos de CPU.

Já vimos em seções anteriores os mecanismos de troca de estados de processos implementados no núcleo do sistema operacional. Observamos que um processo requisitando uma operação de E/S voluntariamente se bloqueia esperando que a operação de E/S seja completada. O processo bloqueado não espera de forma ocupada (*busy wait*), isto é, não fica processando continuamente um *loop* de teste para verificar se a operação de E/S já terminou. Ao contrário, ele libera o processador, e o núcleo do sistema operacional o coloca (coloca seu PCB) na lista de processos bloqueados. O processo permanece "dormindo" até que ele seja "acordado" pelo núcleo do sistema operacional quando sua operação de E/S estiver completa, tirando-o da lista de processos bloqueados e colocando-o na lista de processos prontos para execução.

Operações com semáforos também podem ser implementadas no núcleo do sistema operacional para evitar a espera ocupada (*busy waiting*). Um semáforo é implementado como uma variável protegida e uma fila na qual processos podem esperar por operações V. Quando um processo tenta executar uma operação P em um semáforo cujo valor corrente seja zero, o processo libera o processador e se bloqueia para esperar uma operação V no semáforo. O núcleo do SO coloca o PCB do processo na fila de processos esperando naquele semáforo. Note que alguns sistemas implementam uma fila de espera do tipo FIFO, enquanto outros utilizam filas com prioridades ou mesmo outras disciplinas. Em seguida, o núcleo do SO então realoca o processador para o próximo processo pronto para execução.

Em algum momento, nosso processo na fila do semáforo vai chegar à primeira posição da fila. A próxima operação V vai remover o processo da fila do semáforo e colocá-lo na lista de processos prontos. É claro que processos tentando simultaneamente executar operações P e V em um semáforo ganharão acesso exclusivo ao semáforo pelo núcleo do SO.

Vale notar o caso especial de que em sistemas com um único processador, a indivisibilidade de P e V pode ser garantida simplesmente desabilitando interrupções enquanto operações P e V estão manipulando o semáforo. Isto previne que o processador seja "roubado" até que a manipulação do semáforo esteja completa. Neste momento as interrupções poderão ser novamente habilitadas.

No núcleo de um sistema multiprocessado, a um dos processadores pode ser dada a tarefa de controlar a lista de processos prontos e determinar que processadores executam quais processos. Uma outra abordagem para implementar o núcleo para um sistema multiprocessado é controlar o acesso (via *busy waiting*) a uma lista de pronto compartilhada. Um núcleo de um sistema distribuído poderia ter um proces-

sador controlando a lista de pronto, mas normalmente cada processador gerenciaria sua própria lista de pronto. Portanto, cada processador essencialmente possui seu próprio núcleo. Conforme um processo migra entre os vários processadores em um sistema distribuído, o controle daquele processo é passado de um núcleo para outro.

4.3.8 Monitores

A comunicação interprocessos utilizando semáforos e contadores de eventos parece ser a solução definitiva. Entretanto, se analisarmos mais diretamente estas técnicas, veremos que elas possuem alguns problemas:

São tão primitivos que é difícil expressar soluções para problemas de concorrência mais complexos; seu uso em programas concorrentes aumenta a já difícil tarefa de provar a corretude de programas;

o mau uso dessas primitivas, tanto de forma acidental como maliciosa, poderia corromper a operação do sistema concorrente.

Particularmente com semáforos, alguns outros problemas podem ocorrer:

- Se a operação P for omitida, não é garantida a exclusão mútua;
- se a operação V for omitida, tarefas esperando em uma operação P entrariam em *deadlock*;
- uma vez que a operação P é usada e o processo fica nela bloqueado, ele não pode desistir e tomar um curso de ação alternativo enquanto o semáforo estiver em uso;
- um processo só pode esperar em um semáforo por vez, o que pode levá-lo a *deadlock* em algumas situações de alocação de recursos.

Assim, podemos perceber que o programador deve ser extremamente cuidadoso ao utilizar semáforos, por exemplo. Um súbito erro e tudo poderá parar de funcionar sem nenhuma explicação. É como programar em linguagem *assembly*, só que pior, porque os erros, condições de corrida, *deadlocks*, e outras formas de comportamento imprevisível não podem ser reproduzidos para testar programas.

Para tornar mais fácil a escrita de programas corretos, Hoare (1974) e Brinch Hansen (1975) propuseram uma primitiva de sincronização de alto nível chamada de **monitor**. Um **monitor** é uma coleção de procedimentos, variáveis, e estruturas de dados que são todos agrupados em um tipo especial de módulo ou pacote. Processos podem chamar os procedimentos em um monitor sempre que o desejarem, mas eles não podem diretamente acessar diretamente as estruturas de dados internas do monitor através de procedimentos declarados fora do monitor. O exemplo abaixo ilustra um monitor escrito em nossa linguagem imaginária, semelhante a Pascal:

```
monitor exemplo;
  var i: integer;
     c: condition; { variável de condição }
```

```
  procedure produtor(x: integer);
  begin
  .
  .
  .
  end;

  procedure consumidor(x: integer);
  begin
  .
  end;
end monitor;
```

Monitores possuem uma importante propriedade que os torna útil para atingir exclusão mútua: somente um processo pode estar ativo em um monitor em qualquer momento. Monitores são uma construção da própria linguagem de programação utilizada, de forma que o compilador sabe que eles são especiais, e pode manipular chamadas a procedimentos dos monitores de forma diferente da qual manipula outras chamadas de procedimentos. Tipicamente, quando um processo chama um procedimento de um monitor, as primeiras instruções do procedimento irão checar se algum outro processo está ativo dentro do monitor. Se isto acontecer, o processo que chamou o procedimento será suspenso até que outro processo tenha deixado o monitor. Se nenhum outro processo estiver usando o monitor, o processo que chamou o procedimento poderá entrar.

Fica a cargo do compilador implementar a exclusão mútua nas entradas do monitor, mas é comum utilizar semáforos binários. Uma vez que o compilador, e não o programador, é quem faz os arranjos para a exclusão mútua, é muito menos provável que alguma coisa dê errado. Em qualquer situação, a pessoa escrevendo o monitor não precisa saber como o compilador faz os arranjos para exclusão mútua. É suficiente saber que ao transformar todas as regiões críticas em procedimentos de monitor, dois processos jamais executarão suas regiões críticas simultaneamente.

Apesar de monitores proverem uma maneira fácil de se conseguir exclusão mútua, como acabamos de dizer, isto não é suficiente. É necessário também que se tenha um meio de fazer os processos bloquearem quando eles não podem prosseguir. Considere o seguinte exemplo da relação produtor-consumidor:

```
program produtor_consumidor;
const N = 100;
var cont: integer;

procedure produtor;
begin
  while true do begin
    produz_item;              { produz um item de dado }
    if (cont = N) then
      suspend;          { se o buffer esta' cheio, entra em suspensao }
    entra_item;               { coloca o item produzido no buffer }
```

```
    cont := cont + 1;
    if (cont = 1) then
      resume (consumidor); { se o buffer estava vazio, acorda o consumidor}
  end;
end;

procedure consumidor;
begin
  while true do begin
    if (cont = 0) then
      suspend;          { se o buffer esta' vazio, entra em suspensao }
    remove_item;              { remove o item do buffer }
    cont := cont - 1;
    if (cont = N - 1) then
      resume (produtor);  { se o buffer estava cheio, acoda o produtor }
    consome_item;             { imprime o item }
  end;
end;

begin
  cont := 0;
  parbegin
    produtor;
    consumidor;
  parend;
end.
```

No exemplo acima utilizamos as chamadas de sistema **suspend** e **resume**. Quando um processo percebe que não poderá continuar processando, pois depende de outra condição para continuar, ele se auto suspende através de um **suspend**. Quando o outro processo percebe que já existe condição para o processo suspenso continuar, ele o acorda com um **resume**.

Analisando o exemplo, podemos perceber que ali existe uma condição de corrida. Ela pode ocorrer porque o acesso à variável **cont** é irrestrito. A seguinte situação poderia ocorrer. O *buffer* está vazio e o consumidor acabou de ler o valor de **cont** para checar se era 0. Neste exato momento, o escalonador decide parar a execução do consumidor e começa a executar o produtor. O produtor entra com um item no *buffer*, incrementa **cont**, e percebe que agora **cont** vale 1. Percebendo que **cont** era anteriormente 0, e que portanto o consumidor deveria estar suspenso, o produtor chama **resume** para acordar o consumidor.

Mas o consumidor na verdade não estava suspenso, e o sinal de **resume** é perdido. Quando o consumidor novamente ganha o processador, ele testa o valor de **cont** que ele já havia lido como sendo 0, e então chama **suspend**, entrando em estado suspenso. Mais tarde o produtor irá completar todo o *buffer* e também chamará **suspend**, ficando ambos os processos suspensos para sempre.

Poderíamos resolver este problema utilizando monitores, simplesmente colocando os testes para *buffer* cheio e *buffer* vazio dentro de procedimentos de monitor, mas como o produtor poderia se bloquear se encontrasse o *buffer* cheio?

A solução recai na introdução do conceito de **variáveis de condição**, juntamente com duas operações sobre elas, **wait** e **signal**. Quando um procedimento de monitor descobre que ele não pode continuar (por exemplo, o produtor encontra o *buffer* cheio), ele executa um **wait** em alguma variável de condição, digamos, cheio. Esta ação faz com que o processo que chamou o procedimento bloqueie. Isto também permite que outro processo que tenha sido anteriormente proibido de entrar no monitor possa agora entrar.

Este outro processo, por exemplo, o consumidor, pode acordar seu parceiro suspenso executando um **signal** na variável de condição na qual seu parceiro está esperando. Para evitar que dois processos estejam ativos no monitor ao mesmo tempo, é preciso uma regra que diga o que acontece após um **signal**. Hoare propôs deixar o processo recém-acordado prosseguir, suspendendo o outro. Brinch Hansen propôs que o processo que chame **signal** deve deixar o monitor imediatamente. Em outras palavras, uma chamada a **signal** deve aparecer somente como o último comando em um procedimento de monitor. Utilizaremos a proposta de Brinch Hansen por ser conceitualmente mais simples e também mais fácil de implementar. Se um comando **signal** é executado em uma variável na qual vários processos estão esperando, somente um deles, determinado pelo escalonador do sistema, é acordado.

Um esqueleto do problema produtor-consumidor utilizando monitores é apresentado a seguir:

```
monitor ProdutorConsumidor;
  var cheio, vazio: condition; { variáveis de condição }
      cont: integer;

  procedure colocar;
  begin
    if cont = N then
      wait(cheio);
    entra_item;
    cont := cont + 1;
    if cont = 1 then
      signal(vazio);
  end;
  procedure remover;
  begin
    if cont = 0 then
      wait(vazio);
    remove_item;
    cont := cont - 1;
    if cont = N - 1 then
      signal(cheio);
  end;
```

```
  count := 0;
end monitor;

procedure produtor;
begin
  while true do begin
    produz_item;
    ProdutorConsumidor.colocar;
  end
end;

procedure consumidor;
begin
  while true do begin
    ProdutorConsumidor.remover;
    consome_item;
  end
end;
```

Pode parecer que as operações **wait** e **signal** pareçam similares às operações **suspend** e **resume**. De fato, elas são muito similares, mas com uma diferença crucial: **suspend** e **resume** falharam porque enquanto um processo tentava ficar suspenso, o outro tentava acordá-lo. Com monitores, isso não pode acontecer. A exclusão mútua automática nos procedimentos do monitor garante isso pois, se por exemplo, o produtor dentro de um procedimento do monitor descobre que o *buffer* está cheio, ele será capaz de completar a operação **wait** sem se preocupar com a possibilidade de que o escalonador venha a dar o processador ao consumidor antes que a operação **wait** complete. O consumidor não poderá nem sequer entrar no monitor antes que a operação **wait** tenha terminado e o processo produtor tenha sido marcado como suspenso pelo sistema.

Ao fazer a exclusão mútua de regiões críticas automática, monitores fazem com que a programação paralela seja muito menos sujeita a erros do que com semáforos. Entretanto, monitores possuem suas desvantagens. Como já foi dito, monitores são um conceito da linguagem de programação. O compilador deve reconhecê-los e arranjar a exclusão mútua de acordo. C, Pascal e outras linguagens não possuem monitores, portanto não se pode esperar que os compiladores dessas linguagens dêem algum suporte a exclusão mútua. De fato, como poderia o compilador saber quais procedimentos estão em monitores e quais não estão?

Estas mesmas linguagens não possuem semáforos, mas adicioná-los a elas é fácil: basta escrever algumas rotinas para as operações P e V e adicionar estas rotinas às bibliotecas de funções. Os compiladores nem precisam “saber” do fato que os semáforos existem. É claro que o sistema operacional precisa suportar semáforos, para que as operações P e V possam ser implementadas.

Para utilizar monitores, é necessária uma linguagem que os tenha por natureza. Muito poucas linguagens, como **Concurrent Euclid** os possuem, e compiladores para

elas são raros.

Outro problema com monitores, e também com semáforos, é que eles foram projetados para sistemas monoprocessados, ou para sistemas multiprocessados com memória compartilhada. Em sistemas distribuídos, onde cada CPU possui sua própria memória e está conectada às outras através de um rede de computadores, semáforos e monitores não podem ser aplicados.

4.4 *Deadlocks* e Adiamento Indefinido

Um processo em um sistema multiprogramado é dito estar em uma situação de *deadlock* quando ele está esperando por um evento particular que jamais ocorrerá. Em um *deadlock* em um sistema, um ou mais processos estão em *deadlock*. Em sistemas multiprogramados, o compartilhamento de recursos é uma das principais metas dos sistemas operacionais. Quando recursos são compartilhados entre uma população de usuários, e cada usuário mantém controle exclusivo sobre os recursos particulares a ele alocados, é possível que haja a ocorrência de *deadlocks* no sentido em que alguns usuários jamais sejam capazes de terminar seu processamento.

O estudo de *deadlocks* envolve quatro áreas: prevenir, evitar, detectar e recuperar. Também relacionados com *deadlocks* estão os conceitos de adiamento indefinido e *starvation*.

4.4.1 Exemplos de Deadlocks

Deadlocks podem desenvolver-se de várias maneiras. Se um processo recebe a tarefa de esperar pela ocorrência de um determinado evento, e o sistema não inclui provisão para sinalizar aquele evento, então temos um *deadlock* de um processo. *Deadlocks* desta natureza são extremamente difíceis de detectar, pois estão intimamente associados ao código do processo, provavelmente com erros, neste caso.

A maioria dos *deadlocks* em sistemas reais geralmente envolve múltiplos processos competindo por múltiplos recursos. Vejamos alguns exemplos comuns.

4.4.2 Um *Deadlock* de Tráfego

A figura 4.6 ilustra um tipo de *deadlock* que ocasionalmente se desenvolve em cidades. Um certo número de automóveis estão tentando atravessar uma parte da cidade bastante movimentada, mas o tráfego ficou completamente paralisado. O tráfego chegou numa situação onde somente a polícia pode resolver a questão, fazendo com que alguns carros recuem na área congestionada. Eventualmente o tráfego volta a fluir normalmente, mas a essa altura os motoristas já se aborreceram e perderam tempo considerável.

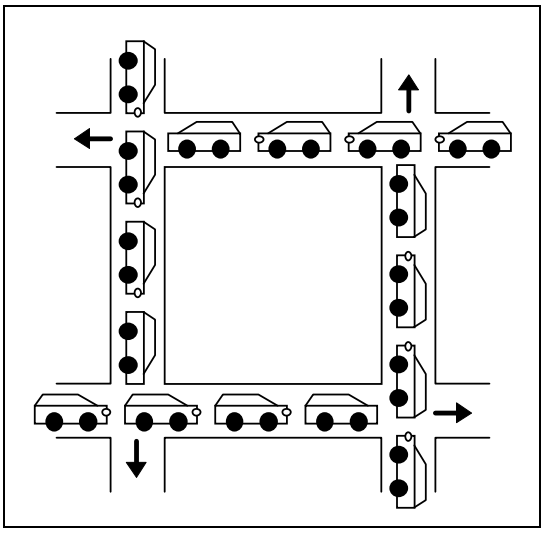


Figura 4.6 - Um *deadlock* de tráfego

4.4.3 Um *Deadlock* Simples de Recursos

Muitos *deadlocks* ocorrem em sistemas de computação devido à natureza de recursos dedicados (isto é, os recursos somente podem ser usados por um usuário por vez, algumas vezes sendo chamados de recursos serialmente reusáveis).

Suponha que em um sistema o processo A detém um recurso 1, e preciso alocar o recurso 2 para poder prosseguir. O processo B, por sua vez, detém o recurso 2, e precisa do recurso 1 para poder prosseguir. Nesta situação, temos um *deadlock*, porque um processo está esperando pelo outro. Esta situação de espera mútua é chamada muitas vezes de espera circular (*circular wait*).

4.4.4 *Deadlock* em Sistemas de *Spooling*

Sistemas de *spooling* costumam estar sujeitos a *deadlocks*. Um sistema de *spooling* serve, por exemplo, para agilizar as tarefas de impressão do sistema. Ao invés do aplicativo mandar linhas para impressão diretamente para a impressora, ele as manda para o *spool*, que se encarregará de enviá-las para a impressora. Assim o aplicativo é rapidamente liberado da tarefa de imprimir. Vários *jobs* de impressão podem ser enfileirados e serão gerenciados pelo sistema de *spooling*.

Em alguns sistemas de *spool*, todo o *job* de impressão deve ser gerado antes do início da impressão. Isto pode gerar uma situação de *deadlock*, uma vez que o espaço disponível em disco para a área de *spooling* é limitado. Se vários processos começarem a gerar seus dados para o *spool*, é possível que o espaço disponível para o *spool* fique cheio antes mesmo de um dos *jobs* de impressão tiver terminado de ser gerado. Neste caso, todos os processos ficarão esperando pela liberação de espaço em disco, o que jamais vai acontecer, e portanto gerando uma situação de *deadlock*. A solução neste caso seria o operador do sistema cancelar um dos *jobs* parcialmente gerados.

Para resolver o problema sem a intervenção do operador, o SO poderia alocar uma área maior de *spooling*, ou a área de *spooling* poderia ser variável dinamicamente.

Alguns sistemas, como o do Windows 3.x/95, utilizam todo o espaço em disco disponível Entretanto, pode acontecer de o disco possuir pouco espaço e o problema ocorrer da mesma forma.

A solução definitiva seria implementar um sistema de *spooling* que começasse a imprimir assim que algum dado estivesse disponível, sem a necessidade de se esperar por todo o *job*. Isso faria com que o espaço fosse sendo liberado gradualmente, na velocidade em que a impressora conseguisse consumir os dados. Este é um problema típico de uma relação produtor-consumidor.

Nos sistemas operacionais mais populares, como o Windows 95, o sistema de *spool* pode ser configurado para começar a imprimir assim que uma página estiver disponível em disco (isto porque algumas impressoras são orientadas a página, como as impressoras *laser*). No Windows 3.x, o *spool* só iniciava a impressão após todo o *job* de impressão ter sido gerado.

4.4.5 Adiamento Indefinido

Em sistemas onde processos ficam esperando pela alocação de recursos ou pelas decisões de escalonamento, é possível que ocorra adiamento indefinido também chamado de bloqueamento indefinido ou *starvation*).

Adiamento indefinido pode ocorrer devido às políticas de escalonamento de recursos do sistema. Quando recursos são alocados segundo um esquema de prioridades, é possível que um determinado processo espere indefinidamente por um recurso conforme processos com prioridades mais altas venham chegando. Os sistemas operacionais devem ser justos com processos em espera, bem como devem considerar a eficiência de todo o sistema. Em alguns sistemas, o adiamento indefinido pode ser evitado permitindo que a prioridade de um processo em espera cresça conforme ele espera por um recurso. Isto é chamado de *aging* (envelhecimento).

4.4.6 Conceitos de Recursos

Um sistema operacional pode ser visto de forma mais ampla como um gerenciador de recursos. Ele é responsável pela alocação de vários recursos de diversos tipos. A riqueza dos tipos de recursos é que faz o assunto Sistemas Operacionais ser interessante de ser estudado.

Alguns recursos são “preemptíveis”. O maior exemplo é a CPU. Memória também pode ser preemptível (veremos memória virtual).

Certos recursos são não-preemptíveis, e não podem ser tomados de processos aos quais foram alocados. Ex. unidades de fita. Contra-ex. disco.

Alguns recursos são compartilhados entre vários processos. Unidades de disco são compartilhadas em geral. Memória principal e CPU são compartilhadas; apesar de que em um instante a CPU pertence a um único processo, mas sua multiplexação entre os vários processos transmite a idéia de que está sendo compartilhada.

Dados e programas são certamente os recursos que mais precisam ser controlados e alocados. Em sistemas multiprogramados, vários usuários podem querer simultaneamente usar um programa editor. Seria desperdício de memória ter-se uma cópia do editor para cada programa executando. Ao contrário, uma única cópia do código é trazida para a memória e várias cópias dos dados são feitas, uma para cada usuário. Uma vez que o código está em uso por vários usuários simultaneamente, ele não pode mudar. Código que não pode ser mudado enquanto está em uso é dito reentrante. Código que pode ser mudado mas que é reinicializada cada vez que é usado é dito serialmente reusável. Código reentrante pode ser compartilhado entre vários processos, enquanto código serialmente reusável só pode ser alocado a um processo por vez.

Quando chamamos recursos compartilhados particulares, devemos ser cuidadosos para determinar se eles podem ser usados por vários processos simultaneamente, ou se podem ser usados por vários processos, mas um de cada vez. Estes últimos são os recursos que tendem estar envolvidos em *deadlocks*.

Um sistema possui um número finito de recursos para serem distribuídos entre processos concorrentes. Os recursos são classificados segundo vários tipos, sendo que cada tipo pode consistir de uma quantidade de instâncias idênticas. Por exemplo, se considerarmos o tipo de recurso CPU, em uma máquina com dois processadores, temos duas instâncias do recurso CPU.

Se um processo requisita uma instância de um tipo de recurso, a alocação de qualquer instância daquele tipo irá satisfazer a requisição. Se em um determinado sistema esta satisfação não ocorrer, isto significa que as instâncias não são idênticas, e que as classes de tipos de recursos não estão definidas corretamente. Como exemplo (Silberschatz, 1994), suponha que um sistema possui duas impressoras. Elas poderiam ser definidas como instâncias de um mesmo tipo de recurso. Entretanto se uma estivesse instalada no andar térreo do prédio, e a outra no 9º andar, os usuários do andar térreo podem não enxergar as duas impressoras como sendo equivalentes. Neste caso, classes de recursos separadas precisariam ser definidas para cada impressora.

Um processo pode requisitar um recurso antes de usá-lo, e deve liberá-lo depois de seu uso. Um processo pode requisitar quantos recursos precisar para desempenhar a tarefa para a qual foi projetado. Obviamente, o número de recursos requisitados não pode exceder o número total de recursos disponíveis no sistema. Em outras palavras, o processo não pode requisitar três impressoras se o sistema somente possui duas.

Em uma situação de operação normal, um processo pode utilizar um recurso somente nesta seqüência:

Requisitar: se a requisição não pode ser atendida imediatamente (por exemplo, o recurso está em uso por outro processo), então o processo requisitante deve esperar até obter o recurso;

Usar: O processo pode operar sobre o recurso (por exemplo, se o recurso é uma impressora, ele pode imprimir);

Liberar: O processo libera o recurso.

4.4.7 Quatro Condições Necessárias para *Deadlock*

- Coffman, Elphick, e Shosani (1971) enumeraram as seguintes quatro condições necessárias que devem estar em efeito para que um *deadlock* exista.
- Processos requisitam controle exclusivo dos recursos que eles necessitam (condição "*mutual exclusion*");
- Processos detêm para si recursos já alocados enquanto estão esperando pela alocação de recursos adicionais (condição "*hold and wait*", ou "*wait for*");
- Recursos não podem ser removidos dos processos que os detêm até que os recursos sejam utilizados por completo (condição "*no preemption*");

Uma cadeia circular de processos existe de forma que cada processo detém um ou mais recursos que estão sendo requisitados pelo próximo processo na cadeia (condição "*circular wait*").

Como todas as condições são necessárias para um *deadlock* existir, a existência de um *deadlock* implica que cada uma dessas condições estejam em efeito. Como veremos adiante, isto nos ajudará a desenvolver esquemas para prevenir *deadlocks*.

4.4.8 Métodos para Lidar com Deadlocks

Basicamente, há três maneiras diferentes de lidar com *deadlocks*:

- Pode ser usado um protocolo para garantir que em um determinado sistema *deadlocks* **jamais** ocorrerão;
- Pode-se deixar o sistema entrar em um estado de *deadlock* e então tratar da sua recuperação;
- Pode-se simplesmente ignorar o problema, e fingir que *deadlocks* nunca ocorrem. Esta solução é usada pela maioria dos sistemas operacionais, inclusive o UNIX.

Para garantir que *deadlocks* nunca ocorrem, o sistema pode tanto usar um esquema de **prevenir deadlocks**, ou **evitar deadlocks**. A prevenção de *deadlocks* é um conjunto de regras de requisição de recursos que garantem que pelo menos uma das condições necessárias para a ocorrência de *deadlocks* não esteja em efeito. Evitar *deadlocks*, por outro lado, requer que seja fornecida ao sistema operacional informação adicional sobre quais recursos um processo irá requisitar e usar durante sua execução. Com o conhecimento dessa informação, é possível decidir, a cada requisição, se o processo pode prosseguir ou se deve esperar. Cada requisição requer que o sistema operacional considere os recursos atualmente disponíveis, os recursos alocados a cada processo, e as futuras requisições e liberações de cada processo, para que possa decidir se a requisição corrente pode ser satisfeita ou deve ser adiada.

Se não são usadas estratégias de prevenção ou para evitar *deadlocks*, existe a possibilidade de ocorrência destes. Neste ambiente, o sistema operacional pode possuir um algoritmo que consiga determinar se ocorreu um *deadlock* no sistema, além de um algoritmo que faça a recuperação da situação de *deadlock*.

Se um sistema que nem previne, evita, ou recupera situações de *deadlock*, se um *deadlock* ocorrer, não haverá maneira de saber o que aconteceu exatamente. Neste caso, o *deadlock* não detectado causará a deterioração do desempenho do sistema, porque recursos estão detidos por processos que não podem continuar, e porque mais e mais processos, conforme requisitam recursos, entram em *deadlock*. Eventualmente o sistema irá parar de funcionar, e terá que ser reinicializado manualmente.

Apesar do método de ignorar os *deadlocks* não parecer uma abordagem viável para o problema da ocorrência de *deadlocks*, ele é utilizado em vários sistemas operacionais. Em muitos sistemas, *deadlocks* ocorrem de forma não freqüente, como por exemplo, uma vez por ano. Assim, é muito mais simples e "barato" usar este método do que os dispendiosos métodos de prevenir, evitar, detectar e recuperar de situações de *deadlock*. Além disso, podem existir situações em que um sistema fica aparentemente "congelado" sem estar, no entanto, em situação de *deadlock*. Imagine, por exemplo, um sistema rodando um processo em tempo real com a máxima prioridade, ou ainda, um processo rodando em um sistema com escalonador não preemptivo.

4.4.9 Prevenção de *Deadlocks*

Como vimos, para que um *deadlock* ocorra, todas as condições necessárias para ocorrência de *deadlocks*, que listamos anteriormente, devem estar em efeito. Isto quer dizer que se garantirmos que somente uma delas não possa ocorrer, estaremos **prevenindo** a ocorrência de *deadlocks* em um determinado sistema. Examinemos as quatro condições separadamente:

4.4.9.1 Negando a Condição *"Mutual Exclusion"*

Conforme afirmamos acima, a condição "*mutual exclusion*" não deve ser negada, pois dois processos acessando um recurso simultaneamente poderiam levar o sistema a uma situação de caos. Imagine o exemplo de dois processos acessando uma mesma impressora ao mesmo tempo! Uma solução é utilizar um sistema de *spool*, onde um único processo de *spool* acessa a impressora diretamente, e não acessa nenhum outro recurso. Uma vez que os processos não imprimem diretamente, e o processo de *spool* acessa somente o recurso impressora, *deadlocks* não podem ocorrer.

O problema é que nem todos os recursos podem ser alocados via *spooling*. Além disso, o próprio sistema de *spooling* pode levar a situações de *deadlock*, conforme já discutimos.

4.4.9.2 Negando a Condição *"Hold and Wait"*

A primeira estratégia de Havender requer que todos os recursos que um processo

precise devem ser requisitados de uma só vez. O sistema deve liberar os recursos segundo uma política "tudo ou nada". Se todos os recursos que o processo requisitou estão disponíveis, então o sistema pode alocá-los todos de uma vez ao processo, que poderá prosseguir. Se, por outro lado, nem todos os recursos requisitados estão disponíveis, então o processo deve esperar até que todos eles estejam disponíveis. Enquanto o processo espera, entretanto, ele não deve deter nenhum recurso. Assim a condição "*hold and wait*" é negada e *deadlocks* simplesmente não podem ocorrer.

Esta solução parece ser boa mas pode levar a um sério desperdício de recursos. Por exemplo, suponha um programa que lê dados de uma unidade de fita, processa-os por uma hora, e em seguida imprime alguns gráficos em um *plotter*. Uma vez que ambas a unidade de fita e o *plotter* estejam disponíveis, o programa pode prosseguir. O desperdício ocorre porque o *plotter* ficará alocado ao processo durante uma hora antes de ser efetivamente utilizado.

Outro problema é a possibilidade de um processo requisitando todos os seus recursos de uma só vez ficar indefinidamente esperando, se outros processos estiverem usando os recursos que ele deseja com bastante freqüência.
De qualquer forma, esta abordagem evita *deadlocks*.

4.4.9.3 Negando a Condição *"No Preemption"*

Negar a condição de "não preempção" é uma estratégia ainda pior do que a anterior. Para vários recursos, como uma impressora, não é interessante que um processo os perca durante seu uso.

4.4.9.4 Negando a Condição *"Circular Wait"*

A condição "*circular wait*" pode ser eliminada de várias formas. Uma maneira é simplesmente estabelecer uma regra que diga que um processo só pode alocar um único recurso em um dado momento. Se ele precisa de um segundo recurso, deve liberar o primeiro. Para um processo que necessita copiar um arquivo bastante grande para uma impressora (o processo de *spooling*, por exemplo), isto é inaceitável.

Uma estratégia melhor seria utilizar a terceira estratégia de Havender, que determina que todos os recursos devem ser numerados em ordem crescente. Assim, processos podem requisitar recursos sempre que quiserem, mas todas as requisições devem ser em ordem crescente de numeração. Tomando a figura 10.3 como exemplo, um processo poderia requisitar o recurso R1 e em seguida o recurso R3, mas não o inverso.

5 GERENCIAMENTO DE MEMÓRIA

Memória é um importante recurso que deve ser cuidadosamente gerenciado. Apesar de um computador doméstico atual possuir dezenas ou até mesmo centenas de vezes mais memória que um computador IBM 7094 (o maior computador existente no início dos anos 60), o tamanho dos programas de computador tem crescido em uma escala tão grande quanto à da quantidade de memória dos computadores.

A memória sempre foi vista como um recurso caro e por isso merece cuidados para o seu gerenciamento correto. Apesar da queda vertiginosa do preço da memória real, esta ainda é muitas vezes mais cara do que a memória secundária (discos, fitas, etc.).

O componente do sistema operacional responsável pela administração da memória é chamado de **gerenciador de memória**. Seu papel consiste em saber quais partes da memória estão ou não em uso, alocar memória para os processos quando dela necessitam e desalocar quando deixam de usá-la ou terminam, e gerenciar as trocas entre a memória principal e o disco quando a memória principal não é grande o suficiente para conter todos os processos.

Este capítulo visa abordar alguns esquemas de gerenciamento de memória, desde alguns simples até os mais complexos. Sistemas de gerenciamento de memória podem ser divididos em dois grupos: aqueles que movem processos entre memória e disco durante sua execução (paginação e *swapping*), e aqueles que não o fazem. Cada abordagem possui vantagens e desvantagens. A escolha do melhor esquema de gerenciamento de memória depende de vários fatores, especialmente do projeto do *hardware* do sistema. Como veremos mais adiante, vários algoritmos para gerenciamento de memória requerem algum suporte do *hardware*.

5.1 Conceitos Básicos

Como vimos no início do curso, memória é um componente essencial para a operação de sistemas de computação modernos. A memória é um grande vetor de palavras ou *bytes* (o tamanho de um palavra depende de cada máquina), cada qual com seu próprio endereço. A CPU busca instruções do programa em memória de acordo com o valor do registrador contador de programas (*program counter*, que no caso dos microprocesadores Intel x86, é chamado de IP – *Instruction Pointer*, ou seja, ponteiro de instruções). Estas instruções podem causar a busca ou o armazenamento adicionais para endereços específicos de memória.

Tipicamente, um ciclo de execução de uma instrução de processador primeiramente carregará uma instrução da memória para o processador. A instrução será decodificada e pode fazer com que operandos sejam carregados da memória. Após a execução da instrução sobre os operandos (por exemplo, uma soma), resultados eventualmente podem ser armazenados de volta na memória. Vale observar que a unidade de memória da máquina apenas enxerga uma seqüência de endereços de memória; ela não sabe como esses endereços são gerados (se é o contador de instruções, se é uma referência a operandos, etc.), ou o quê são seus conteúdos (se são instruções ou dados). Assim, nossa preocupação não é saber como um progra-

ma determina quais endereços de memória ele precisa acessar, mas sim, em saber qual a seqüência de endereços que o programa executando deseja acessar.

5.1.1 Ligação de Endereços (*Address Binding*)

Normalmente, um programa reside em um disco como um arquivo binário executável. O programa deve ser trazido para a memória e introduzido no sistema como um processo para que possa ser executado. Dependendo do sistema de gerenciamento de memória em uso, o processo pode ser movido entre disco e memória durante sua execução. Uma coleção de processos em disco que estão esperando para serem trazidos para memória para execução, formam uma fila de entrada.

O procedimento normal é selecionar um dos processos da fila de entrada e carregá-lo na memória. Conforme o processo executa, ele acesa instruções e dados da memória. Eventualmente o processo termina, e seu espaço de memória é declarado como disponível.

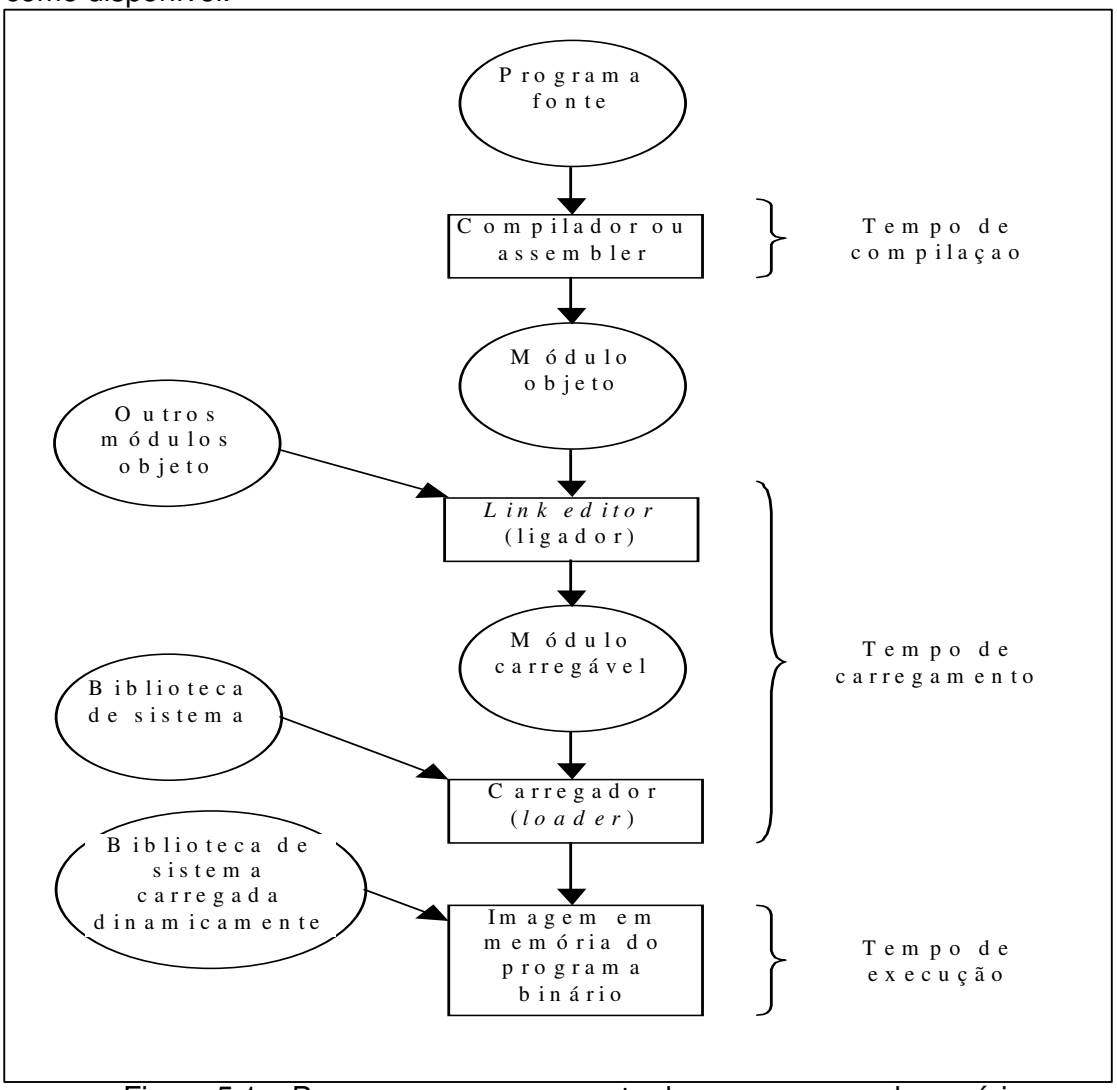


Figura 5.1 – Passos no processamento de um programa de usuário

Muitos sistemas permitem que um processo de um usuário resida em qualquer parte da memória física. Dessa forma, apesar do espaço de endereçamento do computador começar em 00000, por exemplo, o primeiro endereço do processo do usuário não precisa ser 00000. Este arranjo afeta que o programa do usuário pode utilizar. Na maioria dos casos, um programa de usuário passará por várias etapas antes de ser executado, conforme a figura 5.1. Endereços podem ser representados de diferentes formas nestes passos. Endereços no programa fonte normalmente são simbólicos (por exemplo, a variável cont). Um compilador irá tipicamente "ligar" esses endereços simbólicos a endereços relocáveis (como por exemplo, "14 *bytes* a partir do início do bloco desse programa"). O *link editor* ou carregador (*loader*) por sua vez ligará esses endereços relocáveis a endereços absolutos (tal como 74014). Cada uma das ligações é um mapeamento de um espaço de endereçamento para outro.

Classicamente, a ligação de instruções e dados para endereços de memória pode ser feita em qualquer um dos passos citados:

Tempo de compilação: se, em tempo de compilação, é possível saber onde o programa residirá em memória, então código absoluto pode ser gerado. Por exemplo, se é sabido *a priori* que um processo de usuário reside começando na localização R, então o código gerado pelo compilador irá iniciar naquela localização e continuar a partir dali. Se, mais tarde, a localização de início mudar, então será necessário recompilar este código. Os programas em formato .COM do MS-DOS são códigos ligados absolutamente a endereços de memória em tempo de compilação.

Tempo de carregamento: se, em tempo de compilação, não é sabido onde o processo residirá em memória, então o compilador deve gerar código relocável. Neste caso, a ligação final a endereços de memória é adiada até o momento de carga do programa. Se o endereço de início mudar, será somente necessário recarregar o código para refletir a mudança do endereço inicial.

Tempo de execução: se o processo pode ser movido durante sua execução de um segmento para outro, então a ligação a endereços de memória deve ser adiada até o momento da execução do programa. Para este esquema funcionar, é necessário que o *hardware* suporte algumas características, conforme veremos mais adiante.

5.1.2 Carregamento Dinâmico (*Dynamic Loading*)

Para atingir melhor utilização do espaço em memória, pode-se usar carregamento dinâmico. Com carregamento dinâmico, uma rotima não é carregada em memória até que seja chamada. Todas as rotinas são mantidas em disco em um formato relocável. O programa principal é carregado em memória e executado. Quando uma rotina precisa chamar outra rotina, a rotina chamadora primeiramente verifica se a rotina a ser chamada já está carregada. Se não está, o carregador-ligador relocável é chamado para carregar a rotina desejada em memória, e para atualizar as tabelas de endereços do programa para refletir esta mudança. Em seguida, o controle é passado para a rotina recém carregada.

A vantagem do carregamento dinâmico é que uma rotina não usada jamais é carregada em memória. Este esquema é particularmente útil quando grandes quantidades de código são necessárias para manipular casos que ocorrem raramente, como rotinas de tratamento de erros, por exemplo. Neste caso, apesar do tamanho total do programa ser grande, a porção efetivamente usada (e, portanto carregada) pode ser muito menor.

Carregamento dinâmico não requer suporte especial do sistema operacional. É responsabilidade dos usuários projetar seus programas para tomar vantagem deste esquema. Sistemas operacionais, entretanto, podem ajudar o programador provendo rotinas de bilbioteca que implementam carregamento dinâmico.

Sem carregamento dinâmico, seria impossível a uma aplicação atual razoavelmente pesada, como o Microsoft Word, ser carregado rapidamente. Há algumas versões, quando o Microsoft Word é invocado, uma rotina principal de tamanho pequeno rapidamente é carregada, mostrando uma mensagem do aplicativo para o usuário, enquanto as outras rotinas usadas inicialmente terminam de ser carregadas.

5.1.3 Ligação Dinâmica

A figura 3.1 também ilustra bibliotecas ligadas dinamicamente. Muitos sistemas operacionais suportam somente ligações estáticas de bibliotecas de funções, na qual as bibliotecas de sistema são tratadas como qualquer outro módulo objeto e são combinadas pelo carregador dentro da imagem binária do programa em memória. O conceito de ligação dinâmica é similar ao de carregamento dinâmico. Ao invés da carga de rotinas ser adiada até o momento da execução do programa, a ligação de rotinas é adiada até o tempo de execução. Esta característica é normalmente usada com bibliotecas de sistema, tais como bibliotecas de subrotinas da linguagem em uso, como, por exemplo, a biblioteca stdio da linguagem C. Sem essa facilidade, todos os programas em um sistema precisam possuir sua própria cópia da biblioteca básica (ou pelo menos das funções referenciadas pelo programa) incluída na imagem executável. Este esquema desperdiça ambos espaço em disco e espaço em memória principal. Com ligação dinâmica, um *stub* é incluído na imagem para cada referência a uma rotina de biblioteca. Este *stub* é um pequeno código que indica como localizar a rotina de biblioteca apropriada residente em memória, ou como carregar a biblioteca se a rotina ainda não está presente em memória.

Quando este *stub* é executado, ele verifica se a rotina desejada já está em memória. Se a rotina não está em memória, o programa a carrega. De qualquer forma, o *stub* substitui a si mesmo pelo endereço da rotina, e em seguida a executa. Assim, da próxima vez que o trecho de código que referencia a rotina é atingido, a rotina de biblioteca é executada diretamente, sem custo novamente da ligação dinâmica. Sob este esquema, todos os processos que usam uma mesma biblioteca de linguagem executam somente sobre uma cópia do código da biblioteca.

Este recurso pode ser útil também para atualizações de versões de bibliotecas, como o conserto de *bugs*. Uma biblioteca pode ser substituída pela nova versão, e todos os programas que a referenciam usarão a nova versão. Sem ligação dinâmica, todos os programas que usassem tais bibliotecas precisariam ser religados para fun-

cionarem com a nova biblioteca. Para que programas não executem acidentalmente versões mais recentes e incompatíveis de bibliotecas, uma informação sobre versão da biblioteca é incluída tanto no programa quanto na biblioteca. Mais do que uma versão de biblioteca podem estar presentes em memória, e cada programa usa a sua informação de versão para decidir qual biblioteca usar. Pequenas mudanças normalmente mantém o número de versão, enquanto mudanças mais radicais incrementam o número de versão. Este esquema também é chamado de bibliotecas compartilhadas.

Ligação dinâmica precisa da ajuda do sistema operacional para funcionar. Se os processos em memória são protegidos uns dos outros, como veremos adiante, então o sistema operacional é a única entidade que pode verificar se a rotina necessária está no espaço de memória de outro processo, e pode permitir que múltiplos processos acessem os mesmos endereços de memória. Este conceito será expandido quando discutirmos paginação.

Analisando o funcionamento de bibliotecas dinâmicas, vemos que elas utilizam-se de código reentrante para que vários processos compartilhem uma única cópia da biblioteca em memória. Para exemplificar com um sistema operacional conhecido, todos os arquivos .DLL do Microsoft Windows são bibliotecas de ligação dinâmica.

5.1.4 Overlays

Em algumas situações pode ser que a memória física não seja suficiente para conter todo o programa do usuário. Uma forma de resolver esse problema é o uso de *overlays*. As seções do programa não necessárias em um determinado momento podem ser substituídas por uma outra porção de código trazida do disco para execução, conforme ilustra a figura 5.2. O *overlay* manual requer planejamento cuidadoso e demorado. Um programa com uma estrutura de *overlays* sofisticada pode ser difícil de modificar. Alguns compiladores (Turbo Pascal) podem gerenciar *overlays* automaticamente, mas sem eficiência muito grande. Como veremos adiante, os sistemas de memória virtual eliminaram o trabalho do programador com a definição de *overlays*.

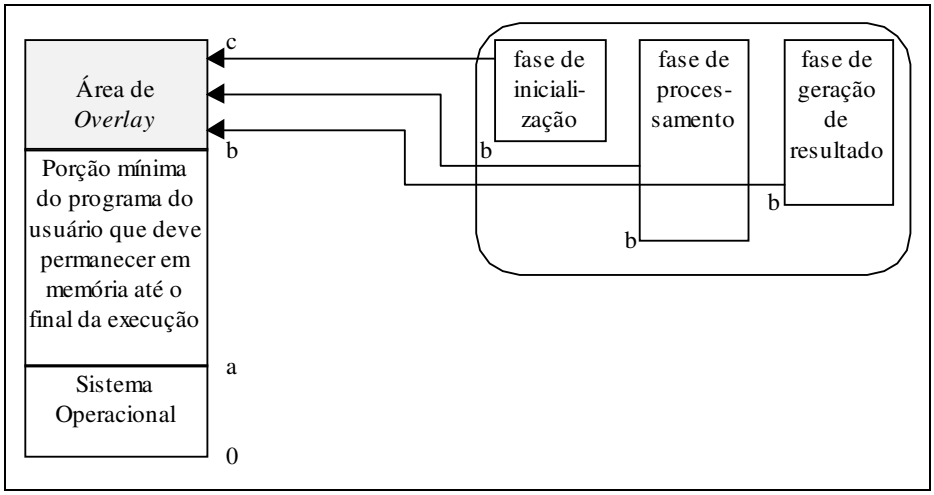


Figura 5.2 - Uma estrutura de *overlay* típica

5.2 Endereçamento Lógico e Endereçamento Físico

Um endereço gerado pela CPU é normalmente referido como sendo um endereço lógico, enquanto que um endereço visto pela unidade de memória do computador (isto é, aquele carregado no registrador de endereço de memória do controlador de memória) é normalmente referido como sendo um endereço físico.

Os esquemas de ligação de endereços em tempo de compilação e em tempo de carregamento resultam em um ambiente onde os endereços lógicos e físicos são os mesmos. Entretanto, a ligação de endereços em tempo de execução faz com que endereços lógicos e físicos sejam diferentes. Neste caso, nos referimos normalmente ao endereço lógico como um endereço virtual. Os termos endereço lógico e endereço virtual são, na verdade, a mesma coisa. O conjunto de todos os endereços lógicos gerados por um programa é chamado de espaço de endereços lógico; o conjunto dos endereços físicos correspondentes a estes endereços lógicos é chamado de espaço de endereços físico. Assim, no esquema e ligação de endereços em tempo de execução, os espaços de endereços lógico e físico diferem.

O mapeamento em tempo de execução de endereços virtuais para endereços físicos é feito pela unidade de gerenciamento de memória (MMU – *Memory Management Unity*), que é um dispositivo de *hardware*. Existem diversas maneiras de fazer este mapeamento, conforme discutiremos mais adiante. Vejamos um esquema de MMU simples (figura 5.3).

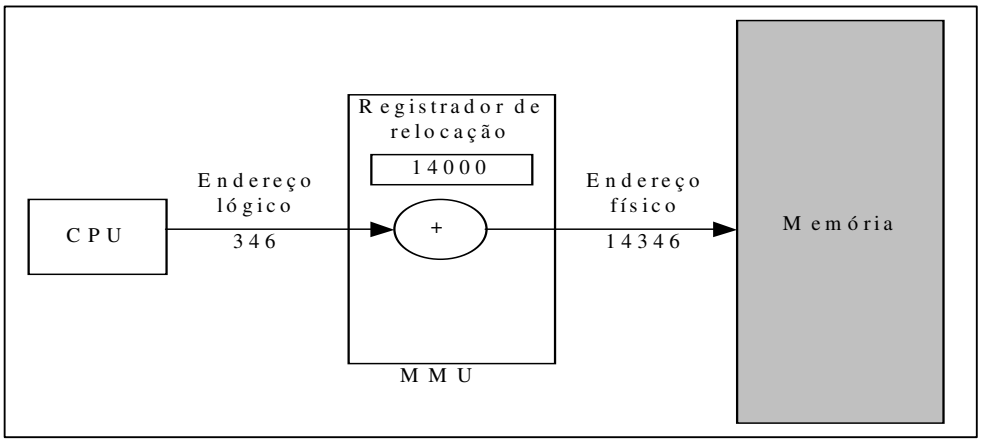


Figura 5.3 – Relocação dinâmica usando um registrador de relocação

Conforme ilustrado na figura, este esquema requer um suporte do *hardware*, particularmente, na MMU. Um registrador de relocação mantém um valor que deve ser somado a qualquer endereço requisitado à memória, gerados por um processo de usuário em execução. Conforme o exemplo, se o endereço base para o processo é 14000, então uma tentativa do processo do usuário acessar o endereço de memória 0 é dinamicamente relocada para o endereço físico 14000, enquanto uma tentativa de acesso ao endereço lógico (virtual) 346 é mapeado para o endereço físico 14346. O sistema operacional MS-DOS, quando executado em um processador Intel 80x86 (uma vez que existem emuladores de MS-DOS para outras máquinas), utiliza quatro registradores de relocação enquanto carrega e executa processos.

Vale observar que o programa do usuário nunca enxerga os reais endereços físicos de memória. O programa em C, por exemplo, pode criar um ponteiro para o endereço 346, armazená-lo em memória, manipulá-lo, compará-lo com outros endereços – sempre como o endereço 346. Somente quando este valor é usado como um endereço de memória, é que ele será mapeado com relação ao endereço base de memória. O programa do usuário lida com endereços lógicos. O *hardware* de mapeamento de memória converte endereços lógicos em endereços físicos. Isto forma a ligação de endereços em tempo de execução que discutimos nos itens anteriores. O localização final de uma referência a um endereço de memória não é determinada até que a referência seja efetivamente feita.

5.3 Swapping

Um processo precisa estar em memória para executar. Um processo, entretanto, pode ser temporariamente ser retirado (*swapped*) da memória para uma área de armazenamento de trocas (área de *swapping*), de forma que mais tarde seja trazido de volta para a memória para que continue executando. Por exemplo, suponha um ambiente multiprogramado com um algoritmo de escalonamento de CPU *round-robin*. Quanto o *quantum* de determinado processo expira, o gerenciador de memória do SO começará a retirar (*swap out*) o processo recém interrompido, e recolocar (*swap in*) outro processo no espaço de memória que foi liberado. Enquanto isso, o escalonador de CPU irá alocar uma fatia de tempo para outro processo em memória. Conforme cada processo tem seu *quantum* expirado, ele será trocado (*swapped*) por outro processo que estava na área de *swapping*. Em uma situação ideal, o gerenciador de memória conseguirá trocar processos em uma velocidade tal que sempre haja processos em memória, prontos para executar, sempre que o escalonador de CPU decidir colocar outro processo em execução. O *quantum* dos processos devem também ser suficientemente grande para que os processos consigam processar por um tempo razoável antes de serem retirados (*swapped out*) da memória.

Uma variação desta política de *swapping* poderia ser usada para algortimos de escalonamento baseados em prioridades. Se um processo com maior prioridade chega no sistema e deseja CPU, então o gerenciador de memória poderia retirar um ou mais processos com prioridades mais baixas de forma que ele possa carregar e executar o processo de prioridade mais alta. Quando este processo de alta prioridade terminasse, os processos de baixa prioridade poderiam ser trazidos de volta para a memória e continuarem executando. Esta variante de *swapping* é às vezes chamada de *roll out, roll in*.

Normalmente, um processo que foi retirado da memória será trazido de volta no mesmo espaço de memória que ocupava anteriormente. Esta restrição pode existir ou não conforme o método de ligação de endereços de memória. Se a ligação de endereços de memória é feita em tempo de compilação ou de carregamento, então o processo não pode ser movido para localizações diferentes de memória. Se a ligação em tempo de execução é usada, então é possível retirar um processo da memória e recolocá-lo em um espaço de memória diferente, uma vez que os endereços físicos são calculados em tempo de execução.

A técnica de *swapping* requer uma área de armazenamento. Normalmente, este espaço para armazenamento é um disco (ou uma partição de um disco) de alta velocidade. Ele deve ser grande o suficiente para acomodar cópias de todas as imagens de memória para todos os usuários, e deve prover acesso direto a essas imagens de memória. O sistema mantém uma fila de pronto (*ready queue*) consistindo de todos os processos cujas imagens de memória estão na área de armazenamento ou em memória real, e que estão prontos para executar. Toda vez que o escalonador de CPU decide executar um processo, ele chama o despachador (*dispatcher*). O despachador verifica se o próximo processo da fila está ou não em memória. Se o processo não está, e não há uma área de memória livre para carregá-lo, o despachador retira (*swap out*) um processo atualmente em memória e recoloca (*swap in*) o processo desejado. Ele então restaura o contexto do processo normalmente (conteúdos de registradores, etc.), muda seu estado para *running*, e transfere o controle para o processo.

É claro que o tempo de troca de contexto no caso de um sistema com *swapping* é razoavelmente alto. Para se ter uma idéia do tempo da troca de contexto, suponhamos que um processo de usuário tenha tamanho de 100KB e que o dispositivo de armazenamento de *swapping* é um disco rígido padrão com taxa de transferência de 1 MB por segundo. A transferência real dos 100KB do processo de ou para memória leva:

100 KB / 1000 KB por segundo = 1/10 segundo = 100 milisegundos
Assumindo que não haja tempo de posicionamento dos cabeçotes do disco rígido, e um tempo médio de latência (tempo até o disco girar para a posição desejada) de 8 milisegundos, o tempo de *swap* leva 108 milisegundos. Como é necessário a retirada de um processo a recolocação de outro, este tempo fica em torno de 216 milisegundos.

Para o uso eficiente de CPU, é desejável que o tempo de execução para cada processo seja longo em relação ao tempo de *swap*. Dessa forma, em um algoritmo de escalonamento *round-robin*, por exemplo, o tempo do *quantum* deveria ser substancialmente maior do que 0.216 segundos.

Note que a maior parte do tempo de *swap* é o tempo de transferência. O tempo de transferência é diretamente proporcional à quantidade de memória trocada (*swapped*). Se um determinado sistema de computação tiver 1 MB de memória principal e um sistema operacional residente de 100 KB, o tamanho máximo para um processo de usuário é de 900 KB. Entretanto, muitos processos de usuários podem ser muito menores que este tamanho, por exemplo 100 KB cada. Um processo de 100 KB poderia ser retirado da memória em 108 milisegundos, ao contrário dos 908 milisegundos necessários para um processo de 900 KB.

Portanto, seria útil saber exatamente quanta memória um processo de usuário está usando, não simplesmente quanto ele poderia estar usando. Assim, precisaríamos apenas retirar da memória a quantidade de memória realmente usada, reduzindo o tempo de *swap*. Para que este esquema seja efetivo, o usuário deve manter o sistema informado de quaisquer mudanças das necessidades de memória.

Dessa forma, um processo com necessidades dinâmicas de memória deverá fazer chamadas de sistema (solicita memória e libera memória) para informar o sistema operacional de suas mudanças de necessidades por memória.

Há outras limitações com o *swapping*. Se desejamos retirar um processo da memória, devemos ter certeza de que ele está completamente ocioso. Uma preocupação em especial é quanto a solicitações de I/O pendentes. Se um processo está esperando por uma operação de I/O, poderíamos querer trocá-lo para a área de *swapping* para liberar a memória que está ocupando. Entretanto, se a operação de I/O está acessando o espaço de memória do usuário de forma assíncrono, como *buffers* de I/O, então o processo não pode ser retirado da memória.

Como exemplo, assuma que a operação de I/O do processo P1 foi enfileirada porque o dispositivo estava ocupado. Se trocarmos o proccesso P1 pelo processo P2 que estava na área de *swapping*, assim que o dispositivo tornar-se livre, a operação de I/O poderá tentar usar uma área de memória que agora pertence ao processo P2. Duas soluções possíveis são: (1) nunca retirar para a área de *swap* um processo com I/O pendente; ou (2) executar operações de I/O somente em *buffers* do próprio sistema operacional. Neste último caso, as transferências entre o sistema operacional e a área de memória do processo somente ocorre quando o processo está presente na memória real.

Atualmente, o *swapping* tradicional é usado em poucos sistemas. Ele requer muito tempo de *swapping* e provê muito pouco tempo de execução para os processos para ser uma solução razoável de gerenciamento de memória. Versões modificadas de *swapping*, entretanto, são encontradas em muitos sistemas.

Uma versão modificada de *swapping* é encontrada em muitas versões de UNIX (mais antigas). Normalmente, o *swapping* é desabilitado, mas é inicializado assim que muitos processos estão rodando, e um certo limiar de quantidade de memória em uso é atingido. Se a carga do sistema reduzir, o *swapping* é novamente desabilitado.

O IBM-PC original (devido ao processador 8086/8088) fornecia pouco suporte do *hardware* para métodos mais avançados de gerenciamento de memória. Os sistemas operacionais da época também tiravam muito pouco proveito do que o *hardware* poderia fornecer em termos de suporte para o gerenciamento de memória. Entretanto, o *swapping* é usado por vários sistemas operacionais em PCs para permitir que vários processos estejam em execução concorrentemente. Um exemplo é o Microsoft Windows, que suporta execução concorrente de processos em memória. Se um novo processo é carregado e a memória é insuficiente, um processo mais antigo é retirado para o disco. Este SO, entretanto, não provê *swapping* completo, uma vez que os processos dos usuários, e não o escalonador, é que decidem se é hora de trocar um processo em memória por outro em disco. Qualquer processo em disco permanece lá até que um processo executando o escolha para execução. A versão mais nova do Windows, o NT (e hoje também, o Windows 95), faz uso dos recursos avançados da MMU encontrada nos PCs de hoje, que foram introduzidos com o processador Intel 80386.

5.4 Alocação Contígua de Memória

A memória principal deve acomodar tanto o sistema operacional como os processos dos usuários. A memória é usualmente dividida em duas partições, uma para o sistema operacional residente, e outra para os processos dos usuários. É possível que o sistema operacional fique alocado tanto no início da memória, quanto no fim. O principal fator que afeta esta decisão é a localização do vetor de interrupções. Como em geral o vetor de interrupções está nas posições de memória mais baixas, é comum colocar o SO no início da memória. A figura 3.4 ilustra possíveis configurações de posicionamento do SO.

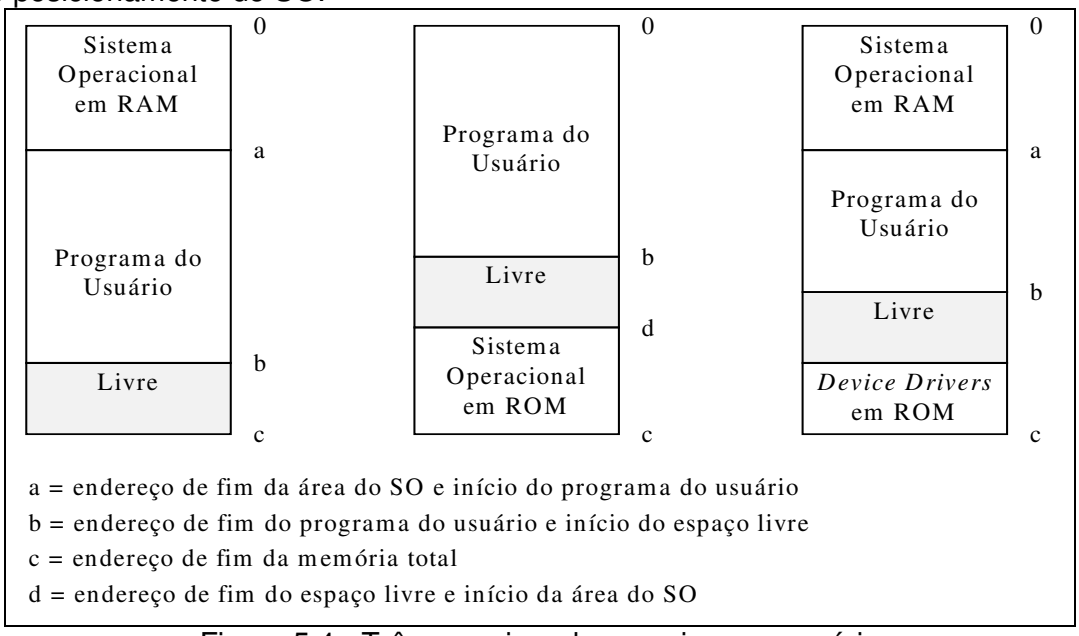


Figura 5.4 - Três maneiras de organizar a memória

O terceiro esquema da figura 3.4 apresenta algumas rotinas em ROM. Esta é a configuração básica do IBM-PC. Alguns sistemas operacionais mais modernos podem completamente ignorar as rotinas em ROM (BIOS – *Basic Input Output System* – Sistema Básico de Entrada e Saída) do PC, que são usadas apenas durante o processo de *boot* do sistema operacional.

5.4.1 Alocação com Partição Única

O esquema mais simples de gerenciamento de memória possível é ter apenas um processo em memória a qualquer momento, e permitir que ele use toda a memória disponível. Os sistemas mais antigos da década de 60 deixavam a máquina sob controle exclusivo do programa do usuário, que era carregado de uma fita ou de um conjunto de cartões (mais tarde de um disco). Esta abordagem não é mais usada, mesmo nos computadores pessoais mais baratos, porque desse modo cada programa deve conter os *device drivers* para todos os dispositivos que ele irá acessar.

Quando o sistema está organizado desta forma, somente um processo pode estar rodando por vez. O usuário digita um comando no console, e o sistema operacional carrega do disco e executa o programa desejado. Quando o programa termina, o sistema operacional novamente apresenta um *prompt* para o usuário, esperando por

um novo comando. A execução de um novo programa irá sobrepôr o conteúdo da memória que foi usada pelo programa anterior.

Apesar de aparentemente não existirem problemas em sistemas monoprogramados, não se deve esquecer que o usuário tem controle total da memória principal. A memória é dividida entre uma porção contendo as rotinas do SO, outra com o programa do usuário, e outra porção não usada. A questão de proteção é quanto à possibilidade do programa do usuário escrever sobre as regiões de memória do SO. Se isso acontecer, ele pode destruir o SO. Se isso for fatal para o programa e o usuário não puder continuar a execução, então ele perceberá algo errado, corrigirá o programa e tentará novamente. Nessas circunstâncias, a necessidade de proteção para o SO não é tão aparente.

Mas se o programa do usuário destruir o SO de forma menos drástica, como por exemplo, a mudança de certas rotinas de I/O, então o funcionamento das rotinas pode ficar completamente às avessas. O *job* continuará rodando. Se os resultados não puderem ser analisados em tempo de execução, um grande tempo de máquina será desperdiçado. Pior ainda, algumas vezes o funcionamento errado não é facilmente perceptível, levando a crer que a execução do *job* está correta. Uma situação ainda mais crítica seria o SO ser modificado de forma que suas rotinas acabem por destruir o sistema como um todo, como no caso de uma rotina de acesso a disco acabar escrevendo em regiões de disco erradas, escrevendo sobre o próprio código do SO no disco, ou ainda destruir informações vitais do disco (tabelas de alocação, partições, etc.).

Está claro que o SO deve ser protegido do usuário. Proteção pode ser implementada simplesmente através de um recurso chamado *boundary register* (registrador de limite), existente em algumas CPUs, conforme ilustrado na figura 5.5.

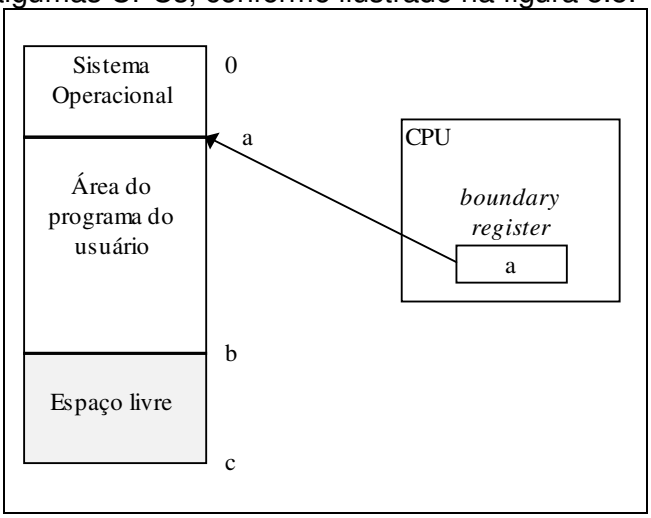


Figura 5.5 - Proteção de memória em sistemas monousuário

O *boundary register* contém o endereço da memória mais alta usada pelo SO. Cada vez que um programa do usuário faz referência a um endereço de memória, o registrador de limitação é verificado para certificar que o usuário não está prestes a escrever sobre a área de memória do SO. Se o usuário tenta entrar no código do SO, a

instrução é interceptada e o *job* terminado com uma mensagem de erro apropriada. Entretanto, é claro que o programa do usuário eventualmente precisa chamar certas funções que estão no código do SO. Para isso o usuário usa uma instrução específica com a qual ele requisita serviços do SO (uma instrução de chamada em modo supervisor - *supervisor call*). Por exemplo, o usuário desejando ler da unidade de fita vai disparar uma instrução requisitando a operação ao SO, que executará a função desejada e retornará o controle ao programa do usuário.

Entretanto, conforme SOs ficaram mais complexos e tornaram-se multiprogramados, foi necessário implementar mecanismos mais sofisticados para proteger as regiões de memória dos usuários uns dos outros. O registrador de limitação (*boundary register*) pode ser implementado através do registrador de relocação, conforme vimos anteriormente. Uma vez que o sistema de memória da máquina sempre assume que o valor do registrador de relocação corresponde ao endereço de memória lógico 0 visto pelos programas, não há necessidade de um registrador de limitação que aponte para o último endereço de memória do SO.

Com a adição de mais um registrador, de limite superior, podemos garantir ainda que um processo de usuário não sobrescreva áreas de memória de outros processos. Assim, o registrador de relocação contém o menor endereço de memória que o processo pode acessar, e o registrador de limitação contém a faixa de endereços (quantidade) que o processo pode endereçar a partir do registrador de relocação. Essa situação é mostrada na figura 5.6.

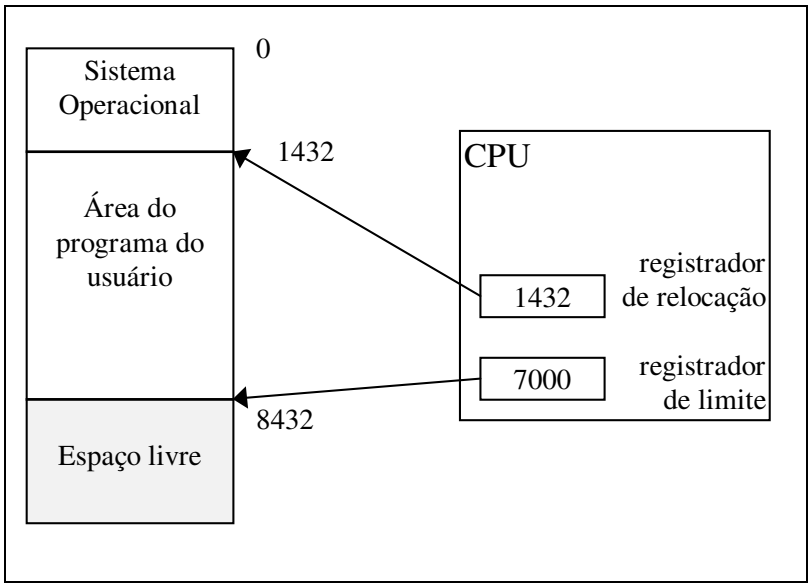


Figura 5.6 – Suporte do *hardware* para registradores de relocação e limite

Com este esquema, toda vez que um processo solicita acesso à memória, ele deve passar pela verificação dos dois registradores, conforme a figura 5.7:

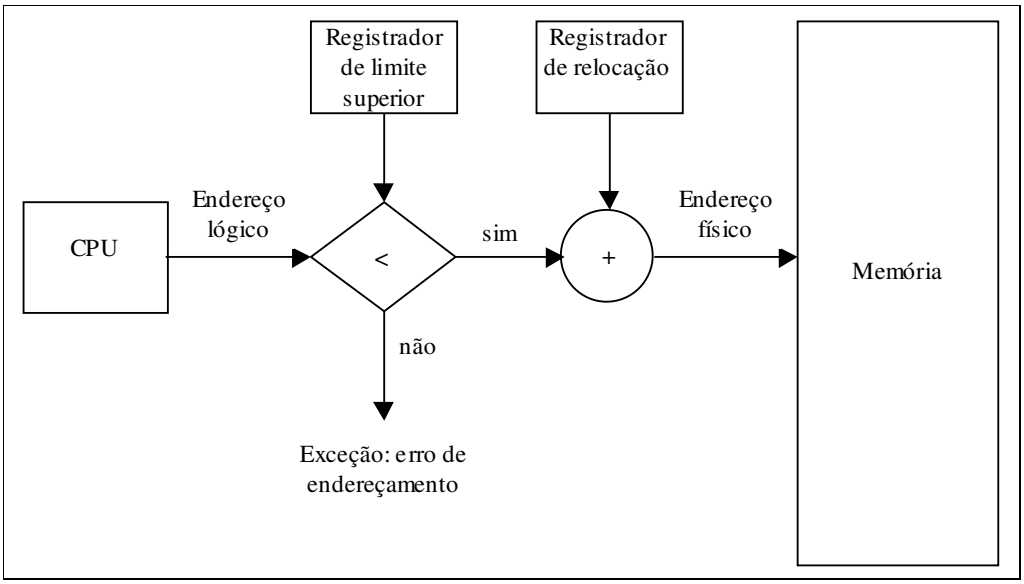


Figura 5.7 – Funcionameto do *hardware* para os registradores de relocação e limite

Quando o escalonador de CPU seleciona outro processo para execução na fila de *ready*, o despachador carrega os registradores de relocação e de limite superior com os valores corretos, da mesma forma que ele o faz com os demais registradores durante a troca de contexto. Como todo endereço de memória gerado pela CPU é verificado com os valores dos registradores, ambos o sistema operacional e os processos dos outros usuários estão protegidos das tentativas do processo executando tentar modificar suas respectivas áreas de memória.

Vale notar que o registrador de relocação fornece um esquema eficiente para permitir que o tamanho do SO em memória cresça ou diminua dinamicamente. Esta flexibilidade é desejável em muitas situações. Por exemplo, o SO contém código e *buffers* para os *device drivers*. Se um *device driver* (ou um outor serviço do SO) não é usado com freqüência, é interesante que suas áreas de código e dados sejam liberadas para uso pelos programas dos usuários. Este tipo de código é chamado de código transiente (ao contrário de residente), pois ele é carregado e descarregado da memória conforme sua necessidade. Dessa forma, ao carregar um *driver* deste tipo, o sistema operacional muda de tamanho em memória, e o registrador de relocação resolve o problema.

Um exemplo de sistema operacional moderno que possui *drivers* carregados e descarregados da memória dinamicamente é o Sun Solaris 2.5. Toda vez que um determinado *driver* (ou até mesmo partes do SO importantes, como os protocolos de rede TCP/IP, etc.) fica um certo tempo sem ser usado por nenhum processo de usuário, ele é automaticamente descarregado da memória. Assim que um processo faz alguma chamada que utiliza este *driver*, o SO automaticamente o carrega. Outro sistema operacional que suporta este tipo de *driver* é o Linux, que incluiu este recurso a partir da versão de kernel 2.0.0.

5.5 Memória Virtual

É uma técnica sofosticada e poderosa de gerência de memória, onde as memórias principal e secundária são combinadas, dando ao usuário a ilusão de existir uma memória muito maior que a memória principal.

O conceito de memória virtual está baseado em desvincular o endereçamento feito pelo programa de endereços físicos da memória principal. Assim, os programas e suas estruturas de dados deixam de estar limitados ao tamnaho da memória física disponível.

5.5.1 Paginação

É a técnica de gerência de memória onde o espaço de endereçamento virtual e o espaço de endereçamento real são divididos em blocos do mesmo tamanho, chamandos de páginas. As páginas no espaço virtual são denomindadas páginas virtuais, enquanto as páginas no espaço real são chamadas de páginas reais ou frames.

Todo mapeamento é realizado em nível de página, através de tabelasde páginas. Cada página do processo virtual possui uma entrada na tabela, com informações de mapeamento que permitem ao sistema localizar a página real correspondente.

Quando um programa é executado,as páginas virtuais são transferidas da memória secundária para a memória principal e colocadas em frames. Sempre que o programa fizer referência a um endereço virtual, o mecanismo de mapeamento localiza, na tabela de processos, o endereço físico do frame.

Nesse sistema, o endereço virtual é formado pelo número da página virtual e um deslocamento dentro da página. O número da página virtual identifica, unicamente, uma página virtual na tabela de páginas, e pode ser considerado como um vetor, enquanto o deslocamento funciona como seu índice. O endereço físico é calculado, então, somando-se o endereço do frame localizado na tabela d epáginas como o deslocamento contido no endereço virtual.

Além da informação sobre a localização da página virtual, a tabela de páginas possui outras informações, dentre elas o bit de validade que indica se uma página está ou não na memória física. Se o bit tem o valor 0 (zero), indica que a página virtual não está na memória principal, enquanto, se for igual a 1 (um), a página está localizada na memória.

Sempre que o processo faz referência a um endereço virtual, o sistema verifica, através do bit de validade, se a página que contém o endereço referenciado está ou não na memória principal. Caso não esteja, o sistema tem de transferir a página da memória secundária para a memória física. Toda vez que o sistema é solicitado para isso, dizemos que ocorreu um page fault (falta de página).

O tamanho da página varia de sistema para sistema, mas normalmente está entre 512bytes a 4 kbytes. A maioria dos estudos em relação ao tamanho ideal da página indica páginas de tamanho pequeno.

5.5.2 Algoritmos de Paginação

A melhor estratégia de relocação de páginas seria, certamente, aquela que escolhesse uma página que não fosse referenciada num futuro próximo, porém, o sistema operacional não tem como prever se uma página será ou não utilizada novamente.

As principais estratégias adotadas pelos sistemas operacionais para relocação de páginas são os seguintes:

• Aleatória ⇒ a escolha aleatória como o nome já diz, não utiliza critéio algum de seleção. Todas as páginas tem a mesma chance de serem selecionadas, inclusive as páginas que são frequentemente referenciadas. Está estratégia consome poucos recursos do sistema, mas é raramente utilizada.

• First-In-First-Out (FIFO) ⇒ nesse esquema, a página que primeiro foi utilizada será a primeira a ser escolhida. Sua implementação é muito simples, sendo necessária apenas uma fila, onde as páginas mais antigas estão no início da fila e as mais recentes no final. Este sistema tende a retornar a mesma página várias vezes.

• Least-Recenty-Used (LRU) ⇒ essa estratégia seleciona a página utilizada menos recentemente, quer dizer, a página que está há mais tempo sem ser referenciada. Ë pouco utilizada devido ao grande overhead causado pela atualização.

• Not-Used-Recenty (NUR) ⇒ a escolha d apágina que não foi recentemente utilizada é bastante semelhante ao esquema LRU. Nessa estratégia exite um flag, que permite ao sistema a implementaçào do algoritmo. O flag de referência indica quando a página foi referenciada ou não, e está associado a cada entrada na tabela de páginas. Inicialmente, todas as páginas estão com o flag indicando que não foram referenciadas. À medida que as páginas são referenciadas, os flags associados a cada página são modificados. Depois de um certo tempo, é possível saber quais páginas foram referenciadas ou não.

• Least-Frequently-Used (LFU) ⇒ nesse esquema, a página menos referenciada, ou seja, a menos frequentemente utilizada, será a página escolhida. Para isso é mantido um contador do número de referências feitas às páginas. A página que tiver o contador com o menor número de referências será a página escolhida, ou seja, o algoritmo privilegia as páginas que são bastante utilizadas. Essa é uma boa estratégia, pois as páginas que entrarem mais recentemente working set serão, justamente, aquelas que estarão como os contadores com menor valor.

5.5.3 Segmentação

É a técnica de gerência de memória, onde os programas são divididos logicamente em sub-rotinas e estruturas de dados, e colocados em blocos de informações na memória. Os blocos tem tamanhos diferentes e são chamados de segmentos, cada um com seu próprio espaço de endereçamento.

A grande diferença entre a paginação e a segmentação é que, enquanto a primeira divide o programa em partes de tamanho fixo, sem qualquer ligação com a estrutura do programa, a segmentação permite uma relação entre a lógica do programa e sua divisão na memória.

O mecanismo de mapeamento é muito semelhante ao da paginação. Os segmentos são mapeados através de tabelas de mapeamento de segmentos e os endereços são compostos pelo número do segmento e um deslocamento dentro do segmento. O número do segmento identifica unicamente uma entrada da tabela de segmentos, onde estão as informações sobre o segmento na memória real. O endereço absoluto é calculado a partir do endereço incial do segmento mais o deslocamento dentro do segmento.

Além do endereço do segmento na memória física, cada entrada na tabela de Segmentos possui informações sobre o tamanho do segmento, se ele está ou não na memória e sua proeção.

O sistema operacional mantém uma tabela com as áreas livres e ocupadas da memória. Quando um novo processo é carregado para a memória, o sistema localiza um espaço livre que o acomode. Na segmentação somente os segmentos referenciados são transferidos da memória secundária para a memória real.

6 BIBLIOGRAFIA

Livro Recomendado:
Fundamentos de Sistemas Opercionais. Editora LTC.
Silberschatz, Galvin, Gagne.

Sistemas Operacionais Modernos. Ed. Prentice Hall do Brasil.
Andrew Tanembaum

Livro de Apoio:
Introdução à Arquitetura de Sistemas Operacionais
Francis B. Machado e Luiz Paulo Maia - Ed. Livros Técnicos e Científicos

Literatura Auxiliar:
Projeto de Sistemas Operacionais em Linguagem C.
Albuquerque, F. Ed. IBM Books EBRAS