Sistemas Operacionais

Conteúdo

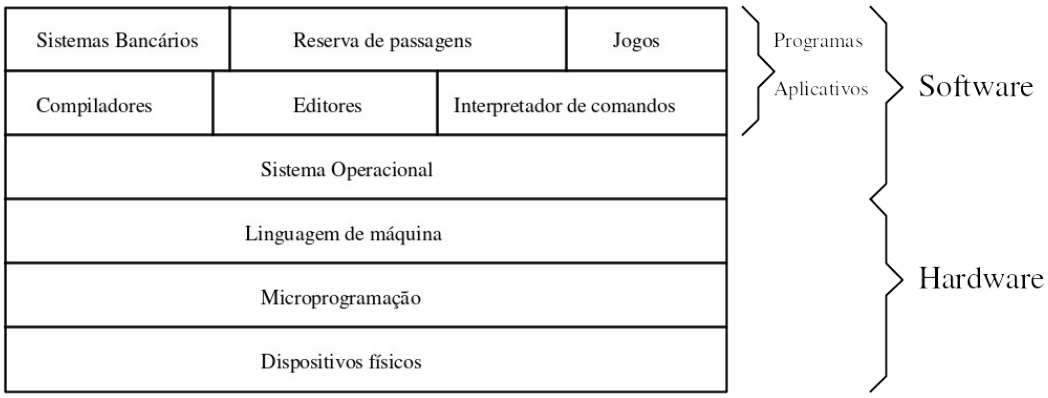
1. Introdução a Sistemas Operacionais

1.1. O que é um Sistema Operacional?

Qualquer pessoa que teve algum contato com um sistema de computação sabe que o mesmo consiste em dois componentes principais: o *hardware* (que engloba toda a parte fisicamente montada, em geral em circuitos eletrônicos) e o *software* (que compreende toda a programação para levar o *hardware* a executar o que foi determinado pelo usuário).

É óbvio que a falta de qualquer um dos componentes acima acarreta a inutilidade do outro, bem como a ineficiência de um tem implicações sérias na utilização do outro.

O que talvez não seja de conhecimento difundido é o fato de que, nos sistemas atuais, tanto o *hardware* como o *software* apresentam uma organização aproximada em camadas, de forma a facilitar a sua utilização. Podemos apresentar um esquema para essa organização como na figura abaixo:



Na parte de *hardware*, a linguagem de máquina é o que o usuário "enxerga" do processador, isto é, o seu único modo de operar com o processador é através da linguagem de máquina. Ao chegar no processador uma instrução de máquina, a microprogramação é a responsável pela interpretação e controle da sua execução, através da ativação dos dispositivos físicos. Em alguns processadores pequenos a interpretação e controle de execução é também realizada por meio de dispositivos, deixando portanto de existir a camada de microprogramação.

Na parte de *software*, o objetivo final são os programas aplicativos. Para possibilitar o desenvolvimento de programas aplicativos existem programas como compiladores, editores de texto, depuradores, entre outros. Estes programas não deixam de ser aplicativos, apenas são voltados ao programador e não ao usuário final. Para permitir a utilização eficiente dos recursos de *hardware*, bem como fornecer facilidades extras ao usuário existe o **sistema operacional** (S.O.).

Podemos dividir as funções do S.O. em duas categorias: a definição de uma **máquina virtual** e o **gerenciamento de recursos**.

1.1.1. S.O. como máquina virtual

Como formador de uma máquina virtual, a função do S.O. é apresentar ao usuário uma máquina com as seguintes características:

a. Facilidade de operação: isto é, o S.O. deve fornecer uma interface entre o usuário e o *hardware* que apresente maior facilidade de programação do que a presente originalmente no *hardware*. Um exemplo típico disto é a escrita em disco rígido. Para esta operação, um controlador precisa das seguintes instruções:

recalibração: corresponde a um ajuste da cabeça de leitura na primeira trilha, pois cada movimento posterior da cabeça é sempre relativo, de forma que se a mesma se apresenta inicialmente mal posicionada, todos os futuros posicionamentos serão errados;

movimento da cabeça: isto é, o deslocamento da mesma para a trilha requerida;

espera do setor: que representa uma espera até que a cabeça de leitura se posicione sobre o setor desejado;

escrita dos dados

verificação: para garantir que os dados foram verdadeiramente escritos e sem nenhum erro

Além deste passos podem existir outros, dependendo do controlador utilizado, do tipo de acesso (leitura, escrita, formatação, etc) e de se ocorrem erros ou não. Atualmente existem diversos controladores de disco que cuidam eles mesmo de tarefas como posicionamento da cabeça, leitura ou escrita, verificação de erros. No entanto mesmo nestes casos o processo de controle restante ao processador central é complicado, pois exige tarefas como:

- controle das trilhas e setores físicos onde se encontra ou deve ser colocada a informação
- tentar novamente no caso de ocorrência de erros, visto que os mesmos são bastante freqüentes nos meios magnéticos atuais devido a condições transitórias

Como vemos, o programador médio não deve ser envolvido com os detalhes deste processo todo. Para isto, os sistema operacionais fornecem métodos muito mais simples e estruturados de uma forma mais voltada aos problemas do usuário do que à estrutura interna do computador

b. Extensão das capacidades da máquina: o S.O. pode fornecer também algumas capacidades não presentes no computador original, como, por exemplo, múltiplos usuários e sistemas de proteção de acesso.

1.1.2. S.O. como gerenciador de recursos

Como um gerenciador de recursos, a função do S.O. é controlar (ou gerenciar) a utilização de todos os recursos fornecidos pelo *hardware* e a sua distribuição entre os diversos programas que competem por ele de forma a garantir:

a. a execução correta dos diversos programas

b. alta eficiência na utilização dos recursos

Dentro deste ponto de vista então, ao S.O. compete cuidar de quem está usando que recurso, aceitar (ordenadamente) requisições de um dado recurso, realizar a contagem de utilização de recursos e mediar conflitos nos pedidos de recursos por vários programas.

1.2. História dos sistemas operacionais

Para uma melhor idéia do curso de desenvolvimento dos sistemas operacionais atuais, apresentamos a esquematização da evolução histórica dos mesmos, enfatizando a relação entre a evolução dos S.O. e os avanços em *hardware*.

1.2.1. Válvulas e painéis de conexão

Os primeiros computadores foram implementados através de válvulas a vácuo, consistindo em salas inteiras de circuito, consumindo energia elétrica suficiente para cidades inteiras. A programação era realizada através de painéis onde as conexões realizadas representavam os 0 e 1 dos códigos binários da linguagem de máquina. Não existia o conceito de sistema operacional, sendo que cada usuário introduzia o seu programa por painéis e aguardava os resultados. A probabilidade de falha do sistema durante a execução de algum programa era altíssima, devido à baixa confiabilidade das válvulas a vácuo.

1.2.2. Transistores e sistemas de lote (batch)

A introdução dos transistores, com a conseqüente redução de tamanho e do consumo e aumento da confiabilidade, permitiu o desenvolvimento dos primeiros sistemas realmente utilizáveis fora dos círculos acadêmicos e governamentais, o que garantiu a comerciabilidade dos mesmos.

Começa a surgir também nesta época a distinção entre projetistas, construtores, operadores, programadores e pessoal da manutenção.

Os computadores ainda eram, entretanto, extremamente grande e caros, devendo ser acondicionados em grandes salas com ar condicionado e operados por pessoal profissional. Seu uso era portanto restrito a entidades governamentais, corporações grandes e universidades.

O processo de execução de uma tarefa (*job*)[1] era, resumidamente:

a. perfuração de um conjunto de cartões com o programa a ser executado
b. o operador pega os cartões e os coloca na leitora. Se o compilador FORTRAN for necessário, ele é colocado (também como um conjunto de cartões) na leitora
c. o resultado sai na impressora e é levado pelo operador para um local onde o usuário o recolhe

Este processo, além de lento, desperdiça muito tempo de programação devido aos deslocamento do operador pela sala, buscando conjuntos de cartões a serem utilizados e pela lentidão dos dispositivos de entrada e saída (leitora de cartões e impressora). Para maximizar a eficiência na utilização dos processadores, e devido ao surgimento das unidades de fita magnética, foi utilizado um novo procedimento:

a. perfuração dos cartões e envio ao operador
b. o operador junta os conjuntos de cartões e, com a utilização de um computador mais barato, grava-os em uma fita magnética
c. a fita magnética é levada ao processador principal e lida
d. os programas da fita são executados e o resultado é gravado em outra fita magnética
e. esta fita de saída é levada ao computador secundário (mais barato), lida e seu conteúdo impresso em uma impressora comum
f. a saída da impressora é entregue aos usuários

[1] Os dois termos ingleses ***job*** e ***task*** são traduzidos como "tarefa". Entretanto, os dois termos são utilizados em sistemas operacionais com sentidos diversos. Por isso utilizaremos preferencialmente os termos ingleses, ou então, o termo português seguido do correspondente inglês.

Este processo garantiu uma maior eficiência na utilização do processador principal, entretanto às custas do tempo de resposta do sistema para cada usuário. Este aumento do tempo de resposta do sistema é devido ao fato de que deve ser juntada uma quantidade razoável de conjuntos de cartões para se gravar uma fita. Desta forma, cada usuário, para obter a resposta a seu programa, deve aguardar a execução de diversos outros programas armazenados na mesma fita. Isto fica mais crítico ainda quando algum dos programas de uma fita apresenta um tempo de execução muito elevado.

1.2.3. Circuitos integrados e multiprogramação

Com a introdução de circuitos integrados, houve uma grande redução no tamanho e custo dos sistemas, bem com um aumento em sua complexidade e generalidade. Isto permitiu o desenvolvimento de dispositivos de entrada e saída inteligentes, de forma que os próprios se responsabilizam pelo controle da transferência de dados entre eles e a memória principal. Outro desenvolvimento importante foi a introdução dos **discos**, que permitem um acesso aleatório à informação contida nos mesmos, diferentemente das fitas magnéticas, que somente permitem um acesso aos dados na ordem em que os mesmos estão gravados (note que isto pode ser escondido através de uma programação cuidadosa, entretanto com alto custo em tempo de execução). Estes foram fatores fundamentais para o sucesso do conceito de multiprogramação, apresentado a seguir.

Simultaneamente com a utilização de circuitos integrados, surgiu o conceito de **multiprogramação**. A idéia provém dos seguintes fatos:

a. durante a execução de programas que se utilizam de muitos cálculos (ex: programas científicos) todos os dispositivos de entrada e saída permanecem inativos
b. durante a execução de programas com alta utilização de entrada e saída (ex: programas comerciais com consultas a base de dados e impressão de relatórios) o processador permanece grande porcentagem do tempo aguardando os dispositivos de entrada/saída.

Desta forma surgiu a idéia de colocar diversas tarefas (*jobs*) dentro de alguma "partição" da memória principal e executando simultaneamente de forma que se alguma precisa aguardar a transferência de dados para um dispositivo, outra tarefa pode utilizar o processador central enquanto isso.

Outro conceito introduzido foi o de "*spooling*" (de *simultaneous periferal operation on line*), que corresponde à leitura imediata dos *jobs* para o disco no momento da sua chegada, sendo que ao terminar um dos *jobs* ativos, um novo *job* é imediatamente carregado do disco para a partição de memória vazia e executado (partição é um trecho de memória alocado a um *job*). Este processo tem a vantagem de que, com a leitura simultânea dos dados para um meio de armazenamento mais rápido e com a transferência de dados entre os meios realizada simultaneamente com a operação da unidade de processamento principal, desapareceu praticamente o tempo manual de montagem e desmontagem de fitas. Além disso, dispondo de diversos *jobs* a serem executados no disco, o sistema operacional podia escolher entre eles por prioridade, e não necessariamente por ordem de chegada.

Entretanto, até este ponto, o sistema continuava sendo um sistema de lotes, sendo o tempo entre a apresentação de um conjunto de cartões e a retirada do resultado extremamente alto principalmente quando se está realizando a depuração de programas. Para diminuir o tempo de resposta do sistema a um dado *job* foi introduzido o conceito de **compartilhamento de tempo** (*time-sharing*), no qual cada usuário possui um terminal ligado em linha com o computador, podendo ainda o computador rodar, no fundo, alguns lotes com a utilização do tempo disponível devido à lentidão de entrada de dados dos usuários.

Nesta época também surgiram os minicomputadores, com uma menor capacidade de processamento numérico mas também um muito menor custo.

Obs: Multiprogramação e multiprocessamento: Estes conceitos devem ser claramente distinguidos, sendo definições possíveis para os mesmos as seguinte:

multiprogramação corresponde a diversos programas distintos executando em um mesmo processador

multiprocessamento corresponde a diversos processadores, dentro de um mesmo sistema de computação, executando programas diversos ou cooperando na execução de um mesmo programa.

Note que foi a existência de multiprocessamento entre os dispositivos de entrada/saída e o processador central que tornou atrativa a introdução da multiprogramação, mas a relação para por aí.

1.2.4. Computadores pessoais e redes

Com a integração em larga escala e o surgimento dos microcomputadores, surge também o conceito de *user-friendly* para S.O., que corresponde ao desenvolvimento de sistemas operacionais para serem utilizados por pessoas sem nenhum conhecimento de computação e que provavelmente não têm nenhum interesse em vir a conhecer algo.

Um outro desenvolvimento interessante que foi bastante impulsionado pelos microcomputadores (apesar de não depender dos mesmos) é o de sistemas operacionais para **redes** de computadores, que consistem em computadores distintos interligados por elementos de comunicação. Os sistemas operacionais para redes são divididos em duas categorias:

a. **sistemas operacionais de rede**: no qual cada usuário tem conhecimento de seu próprio computador e pode acessar dados em outros computadores
b. **sistemas operacionais distribuídos**: em que o sistema operacional faz com que todos os computadores da rede formem uma unidade, de forma que nenhum usuário tem conhecimento de quantos computadores há na rede ou de em qual (ou quais) computador o seu específico programa está executando.

1.4. Conceitos de sistemas operacionais

A interface com o sistema operacional é realizada através de instruções extendidas denominadas **chamadas de sistema** (*system calls*). Veremos agora como são organizadas as chamadas de sistema em UNIX, após o que estudaremos brevemente as principais. Existem duas categorias básicas de chamadas de sistema, uma que trata dos processos e outra que trata do sistema de arquivos.

1.4.1. Processos

Um dos conceitos básicos de qualquer sistema operacional, principalmente os multiprogramados é o de **processo**. Podemos definir um processo como um programa em execução. Ele consiste em:

a. programa executável
b. dados do programa
c. pilha do programa
d. ponteiro de pilha
e. contador de programa
f. outros registradores e informações necessárias para executar o programa.

Para fixar melhor o conceito, pense num sistema de compartilhamento de tempo. Este sistema realiza constantemente a interrupção da execução de um processo e em seguida a sua reiniciação. Para

permitir que um processo seja iniciado no mesmo ponto em que parou, todas as informações necessárias sobre o estado do sistema devem ser guardadas.

Em muitos S.O. as informações sobre cada processo que não estejam contidas em seu espaço de endereçamento são armazenadas em uma tabela chamada **tabela dos processos** (*process table*).

Um processo suspenso consiste então em seu espaço de armazenamento, chamado **imagem de memória** (*core image*) e a sua entrada na tabela de processos.

As mais importantes chamadas de sistema para gerenciamento de processos são as que criam e terminam um processo. Por exemplo, quando o usuário pede a compilação de um programa, o interpretador de comandos (também chamado *shell*) precisa criar um novo processo que irá executar a compilação. Após terminada a compilação, o processo do compilador executa uma chamada de sistema que irá terminá-lo.

Cada processo pode, por sua vez, criar outros processos, chamados **processos filhos** dele. O processo que criou um dado processo é chamado **processo pai** do novo processo.

Outras chamadas de sistema são as que transmitem e controlam **sinais** entre os diversos processos (incluindo-se os processos do S.O.). Quando um sinal é recebido por um processo, o mesmo interrompe o que estiver executando, salva o estado atual, e passa a executar uma rotina que trata do sinal indicado (p. ex.: pode ocorrer um sinal quando a mensagem que um processo enviou para outro não foi recebida dentro de certo tempo, neste caso a rotina que cuida do sinal pode realizar uma tentativa de retransmissão da mensagem perdida). Os sinais são o correspondente em *software* do conceito de interrupção de *hardware*.

Cada processo tem conhecimento do usuário que o criou, através do armazenamento do identificador de usuário, `uid`, de forma a garantir os esquemas de proteção. Existe um usuário, o ***super-user***, que tem privilégios especiais e pode violar muitos esquemas de proteção.

1.4.2. Arquivos

Existem chamadas de sistema para a criação, leitura e escrita de arquivos, entre outras. Estas chamadas fornecem a interface entre o usuário e o *hardware* dos dispositivos de entrada e saída, de forma a apresentar uma organização estruturada e clara dos mesmos, além de garantir as proteções do sistema.

Em UNIX os arquivos são agrupados em **diretórios**, que por sua vez podem conter outros diretórios formando uma árvore. A especificação completa de um arquivo consiste então, além do nome do mesmo, do caminho pela árvore de diretório até chegar a ele (é o *path name*). Nomes de arquivo que não contém o caminho são procurados no **diretório de trabalho** (*working directory*) corrente.

Tanto arquivos como diretórios têm associados a especificação de uma proteção através de um código de nove bits formado do seguinte modo:

```
rwxrwxrwx
```

sendo que cada letra corresponde a um bit e que o primeiro grupo `rwx` corresponde à permissão de acesso pelo proprietário (*user*), o segundo grupo corresponde à permissão de acesso pelo grupo ao qual o usuário pertence (*group*) e o terceiro corresponde à permissão de acesso por todos os outros usuários (*others*). O *super-user* tem acesso livre a todos os arquivos. O significado de cada letra é o seguinte:

`r` : indica permissão de leitura

`w` : indica permissão de escrita

`x` : indica permissão de execução

Quando queremos negar um dado tipo de acesso a um dado tipo de usuário basta substituir a letra correspondente por um hífen "–". Assim a proteção `rwxr-x--x` indica que o proprietário pode ler,

escrever ou executar o arquivo, o grupo ao qual o proprietário pertence pode ler ou executar, mas não alterar ou apagar o arquivo e que todos os outros usuários podem executar mas não ler o conteúdo nem modificá-lo.

Antes de usar qualquer arquivo, o mesmo deve ser aberto, e neste momento a permissão do usuário para sua utilização é checada. Caso o acesso seja permitido, o sistema retorna um código chamado **descritor de arquivo** (*file descriptor*) que será utilizado para identificar o mesmo a partir desse ponto.

Outra operação importante com arquivos é a **montagem**, utilizada em geral para se introduzir unidades externas, como *pendrives* USB no sistema de arquivos para poderem ser acessados. Assim um *pendrive*, que possui uma dada estrutura de arquivos dentro dele, não pode ser utilizado enquanto não for montado. A partir da montagem a sua estrutura de arquivos (possivelmente com diretórios) passa a fazer parte da árvore total do sistema e pode ser acessada. Após a utilização, o *pendrive* deve ser desmontado antes da remoção.

Outro conceito importante é o de **arquivo especial**, que é utilizado para acessar dispositivos de entrada e saída como se fossem arquivos. Assim, um *modem* e uma impressora podem ser utilizados como se fossem arquivos comuns, apenas com algumas restrições de acesso (por exemplo, não podemos ler a impressora ou removê-la do sistema).

Existem três arquivos especiais que são assinalados no momento em que os processos são iniciados:

standard input, com descritor de arquivo 0, correspondendo à entrada de dados pelo terminal do usuário

standard output, com descritor de arquivo 1, correspondendo à saída de dados pelo terminal do usuário

standard error, com descritor de arquivo 2, correspondendo também à saída de dados pelo terminal mas utilizado para mensagens de erro.

Outra possibilidade que indicaremos aqui é a de formação de *pipes*. Um ***pipe*** é uma espécie de pseudo-arquivo utilizado para conectar dois processos entre si, de forma que a saída de dados de um é enviada à entrada de dados do outro (através do *pipe*).

1.5. Chamadas de sistema

Em geral, o processo de utilização do sistema operacional é do seguinte tipo:

1. chamada do sistema por meio de passagem de parâmetros em locais bem definidos (p.ex.: registradores ou pilha)
2. execução de uma instrução especial denominada **chamada do *kernel*** ou **chamada do supervisor**, que modifica o estado da máquina para o modo *kernel* ou modo supervisor
3. busca em uma tabela para verificar qual a rotina específica chamada
4. execução da rotina requisitada
5. volta ao **modo de usuário** e retorno ao programa que chamou.

Veja que aqui consideramos uma máquina que pode operar em dois modos de execução: o modo *kernel* e o modo usuário, sendo que algumas instruções somente podem ser executadas no primeiro modo.

Apresentaremos a seguir as principais chamadas de sistema do UNIX, agrupadas em seis categorias principais de acordo com suas funções.

1.5.1. Chamadas para gerenciamento de processos

`pid=fork()` cria um novo processo igual ao atual

`s=wait(&status)` espera que um processo filho termine e pega seu *status*
`s=execve(name,argv,envp)` troca a imagem de memória de um processo (executa o programa `name`)
`exit(status)` termina um processo e retorna seu *status*
`size=brk(addr)` ajusta o tamanho do segmento de dados para `addr`
`pid=getpid()` devolve o identificador do processo.

Um exemplo de utilização para estas rotinas é o seguinte esqueleto para um interpretador de comandos (*shell*):

```
while (true) {
  le_comando(comando,parametros);
  if (fork() != 0) { /* se for pai */
    wait(&estado);
  }
  else {  /* se for o filho */
    execve(comando, parametros, 0);
  }
}
```

Devemos notar aqui que a chamada `fork()` cria um novo processo idêntico ao atual, mas retorna zero apenas para o processo filho, retornando ao pai o identificador de processo (`pid`) do filho; esta é a razão do teste de `fork` com zero, para que cada processo possa determinar se é pai ou filho.

1.5.2. Chamadas para sinalização

`oldfunc=signal(sig,func)` faz com que um sinal seja capturado, ignorado, etc
`s=kill(pid,sig)` envia um sinal para um processo
`residual=alarm(seconds)` pede um sinal `SIGALRM` após certo período
`s=pause()` suspende o processo até o próximo sinal
Exemplos de sinais fornecidos pelo UNIX são:
`SIGHUP`: detecção pelo *modem* do rompimento de uma conexão
`SIGINT`: processo interrompido via teclado
`SIGQUIT`: sinal de desistência do teclado (sinal gerado por `cntrl-\`)
`SIGILL`: instrução ilegal
`SIGTRAP`: armadilha para depuração de programas
`SIGIOT`: instrução IOT
`SIGEMT`: instrução EMT
`SIGFPE`: *overflow*/*underflow* de ponto flutuante
`SIGKILL`: mata um processo
`SIGBUS`: erro no duto
`SIGSEGV`: violação de segmentação
`SIGSYS`: argumento errado para chamada de sistema
`SIGPIPE`: escrita num *pipe* sem leitura
`SIGALRM`: alarme após dado tempo
`SIGTERM`: sinal de terminação gerado por *software*

1.5.3. Chamadas para gerenciamento de arquivos

`fd=create(name,mode)` cria um novo arquivo (trunca se já existir)

`fd=mknod(name,mode,address)` cria um nó de identificação regular, especial ou de diretório
`fd=open(file,how)` abre um arquivo para leitura, escrita ou ambos
`s=close(fd)` fecha um arquivo aberto
`n=read(fd,buffer,nbytes)` lê dados de um arquivo para *buffer*
`n=write(fd,buffer,nbytes)` escreve dados do *buffer* num arquivo
`pos=lseek(fd,offset,whence)` move o ponteiro de um arquivo para algum ponto no mesmo
`s=stat(name,buffer)` lê e retorna o estado de um arquivo
`s=fstat(fd,buf)` lê e retorna o estado de um arquivo
`fd=dup(fd1)` aloca um novo descritor de arquivo (`fd`) para um arquivo aberto
`s=pipe(&fd[0])` cria um *pipe*
`s=ioctl(fd,request,argp)` realiza operações especiais em arquivos especiais
`mknod` cria arquivos especiais, e só pode ser utilizado pelo *super-user*.
Para exemplificar, vejamos um esqueleto para um *pipeline* de dois processos:

```
#define STD_INPUT 0
#define STD_OUTPUT 1

void pipeline(char *processo1, char *processo2) {
  int fd[2];

  pipe(&fd[0]);   /* cria um pipe */
  if (fork() != 0) {
    close(fd[0]); /* processo 1 não lê do pipe */
    close(STD_OUTPUT);
    dup(fd[1]);   /* faz a saída padrão fd[1] */
    close(fd[1]); /* pipe orig. desnecessário */
    execl(processo1, processo1, 0);
  }
  else {
    close(fd[1]); /* processo 2 não escreve no pipe */
    close(STD_INPUT);
    dup(fd[0]);   /* faz entrada padrão fd[0] */
    close(fd[0]); /* pipe orig desnecessario */
    execl(processo2, processo2, 0);
  }
}
```

Note que ao se realizar um dup, o `fd` de número mais baixo disponível no momento é o utilizado.

1.5.4. Chamadas para gerenciamento de diretório

`s=link(name1,name2)` cria uma nova entrada de diretório, `name2`, para o arquivo `name1`
`s=unlink(name)` remove uma entrada de diretório
`s=mount(special,name,rwflag)` monta um sistema de arquivos
`s=unmount(special)` desmonta um sistema de arquivos
`s=sync()` descarrega todos os blocos que estão na memória *cache* para o disco
`s=chdir(dirname)` muda o diretório de trabalho
`s=chroot(dirname)` muda o diretório raiz

1.5.5. Chamadas para proteção

`s=chmod(name,mode)` muda os bits de proteção associados com um arquivo
`uid=getuid()` retorna o **`uid`** de quem chama
`gid=getgid()` retorna o **`gid`** de quem chama
`s=setuid(uid)` modifica o **`uid`** de quem chama
`s=setgid(gid)` modifica o **`gid`** de quem chama
`s=chown(name,owner,group)` muda o proprietário e grupo de um arquivo
`oldmask=umask(complmode)` muda uma máscara usada para mascarar os bits de proteção na criação de arquivos

1.5.6. Chamadas para gerenciamento de tempo

`seconds=time(&seconds)` vê quanto tempo se passou (em segundos) desde as zero horas de 1 de janeiro de 1970
`s=stime(tp)` modifica o tempo que se passou desde 1 de janeiro de 1970
`s=utime(file,timep)` ajusta o tempo de "último acesso" para um arquivo
`s=times(buffer)` retorna os tempos envolvidos na execução de um processo

1.6. Estrutura do S.O.

Vejamos agora alguns modos como podem ser estruturados os sistemas operacionais.

1.6.1. Sistemas monolíticos

Poderia ser intitulado "A grande confusão", pois corresponde ao sistema não ter estrutura nenhuma. O sistema consiste de diversas rotinas, cada uma das quais pode chamar qualquer uma das outras desde que julgue conveniente. Para isso cada rotina tem uma interface muito bem definida em termos de parâmetros de entrada e parâmetros de saída.

1.6.2. Sistemas em camadas (layers)

Consiste na elaboração do sistema em camadas de forma a que cada uma somente utilize os recursos fornecidos pela camada imediatamente inferior, não se ocupando com a existência de camadas abaixo desta.

Uma generalização deste método de camadas é o de anéis, onde o sistema é organizado em anéis concêntricos, sendo que cada anel mais interno é mais privilegiado que o mais externo. Todas as chamadas para anéis mais internos têm sua validade testada antes da execução.

A organização em camadas é apenas uma ajuda na estruturação do sistema, enquanto que a estruturação em anéis é mantida durante a execução.

1.6.3. Máquinas virtuais

Neste conceito, utilizado basicamente no sistema VM/370, o sistema operacional fornece, a cada usuário, uma máquina virtual idêntica à máquina real, de forma que cada um tem a ilusão de dispor de todo o sistema para ele. Cada uma das máquinas virtuais pode, por sua vez, rodar um sistema operacional completo, independente dos executados em outras máquinas virtuais do mesmo sistema.

1.6.4. Modelo de cliente-servidor

Neste modelo resume-se o sistema operacional ao mínimo possível. O modo usual é formar um *kernel* que se ocupa apenas da comunicação de dados entre processos, chamados clientes e servidores, sendo todo o restante das funções do sistema implementados como programas de usuário. Este processo faz com que cada parte do sistema, por ser isolada, se torne simples e manuseável, impedindo também que a falha em um módulo acarrete falha em todos os outros.

Outra vantagem deste sistema é a facilidade de adaptação a redes de computadores, podendo cada servidor ou cliente ser colocado em máquinas diferentes.

Nem todas as funções do sistema operacional podem, entretanto, ser realizadas por programas no espaço do usuário em geral. Dois modos de tratar com isso são:

1. deixar alguns processos servidores críticos dentro do *kernel*, mas fazendo com que os mesmos se comuniquem com os outros processos através dos mesmos mecanismos dos processos de usuários
2. manter no *kernel* apenas as seções críticas de acesso a dispositivos e deixar todas as decisões políticas para servidores no espaço de usuário.

1.8. Exercícios

1.1. Quais são as duas funções principais de um S.O.?

1.2. O que é multiprogramação e qual a vantagem que a mesma traz com relação à eficiência de utilização dos recursos computacionais?

1.3. Por que compartilhamento de tempo não foi amplamente utilizado na segunda geração de computadores?

1.4. Um arquivo em UNIX, cujo proprietário tem `uid=12` e `gid=1`, tem modo de proteção `rwxr-x---`
Outro usuário com `uid=6` e `gid=1` tenta executar este arquivo. O que acontecerá?

1.5. Para as quatro formas de estrutura de sistemas operacionais, apresente uma vantagem e uma desvantagem.

1.6. O conceito de interrupção é um dos mais importantes para a operação de dispositivos de entrada e saída. Explique, sucintamente, o que você entende por “interrupção”.

1.7. Nos primeiros computadores (bem como em diversos microcomputadores), cada byte de dado lido ou escrito era manuseado diretamente pela UCP. Qual a implicação disto para a multiprogramação?

1.8. Quais das instruções abaixo devem ser permitidas apenas no modo *kernel* (ou executivo, ou supervisor, que é o modo de execução reservado ao S.O.):
(a) desabilita todas as interrupções
(b) lê a data atual

(c) altera a data atual
(d) muda o mapa de memória

1.9. Em vista do fato da existência do *super-user* trazer diversos problemas de segurança, por que tal usuário existe, isto é, por qual razão não se elimina este conceito?

1.10. Por que a tabela de processos é necessária num sistema de compartilhamento de tempo? Ela também é necessária em um computador pessoal em que apenas um processo existe, utilizando toda a máquina para si até terminar?

1.11. O *cache de disco* é um trecho de memória que apresenta um cópia de alguns blocos do disco (aqueles que o sistema, de alguma forma, decide que são os mais importantes no momento; as políticas de decisão e outros detalhes sobre o *cache* serão vistos no decorrer do curso). Desta forma, algum bloco de disco é copiado para o *cache* e todas as leituras e escritas nesse bloco são realizadas no *cache* (um bloco é uma parcela do disco de um tamanho pré definido). Quando existe uma escrita em um bloco que está no *cache*, este passa a ter um conteúdo diferente e mais atualizado que o correspondente bloco em disco; portanto, antes do conteúdo de um bloco em *cache* ser descartado, o correspondente bloco no disco deve ser atualizado. A chamada de sistema `sync` do UNIX realiza a atualização de todos os blocos contidos no cache, guardando seus conteúdos nos correspondentes blocos do disco. Ao ser iniciado o UNIX, um processo, chamado `update` é iniciado, e se encarrega de realizar `sync` a cada 30 segundos. Por que razão tal processo (`update`) é utilizado? Por que não deixar este trabalho somente a cargo dos processos que utilizam escrita no disco?

1.12. Haveria alguma utilidade na utilização da técnica de *spool* em um sistema operacional para um computador que tem dispositivos controlados diretamente pelo processador central (isto é, no qual não existe paralelismo entre computação e acesso às unidades de entrada e saída)? Explique?

1.13. Qual a importância da interrupção para um sistema multiprogramado?

1.14. O conceito de cliente/servidor é muito utilizado em sistemas distribuídos e redes de computadores. Este conceito pode também ser útil em um sistema com um único computador?

2. Processos

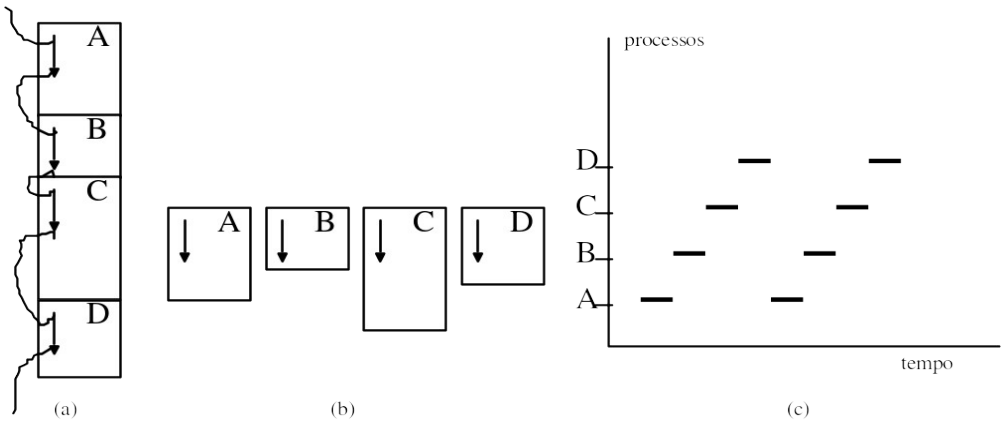
2.1. Introdução a processos

No método tradicional de S.O., consideramos todo o *software* como organizado em um número de processos seqüenciais ou, mais simplesmente, processos.

Definimos um processo como um programa em execução, sendo que para sua especificação completa devemos incluir tanto o programa propriamente dito como os valores de variáveis, registradores, contador de programa (*Program Counter*, PC), e outros dados necessários à definição completa de seu estado.

Cada processo trabalha como se possuísse para si uma UCP (unidade central de processamento, o processador principal do computador) própria, chamada **UCP virtual**. Na realidade, frequentemente uma única UCP é compartilhada por vários processos. Isto é, existe apenas **uma** UCP real, mas **tantas UCP virtuais quantos forem os processos**. Sistemas com **multiprocessamento** apresentam diversas UCPs reais, entretanto mesmo nestes casos ocorre a necessidade de compartilhamento de várias ou todas as UCPs pelos diversos processos.

Os processos, em uma mesma UCP real executam **um por vez**, mas essa execução é realizada de forma a criar a ilusão de que os mesmos estão executando em paralelo, conforme figura apresentada abaixo, onde mostramos a execução na UCP real (a), a aparência criada ao usuário (b), e a distribuição de tempo da UCP real entre os diversos processos (c).



Um fator derivado do método como a implementação das UCPs virtuais é realizada, e que apresenta fundamental importância no desenvolvimento de sistemas operacionais, é que a taxa de execução de cada processo **não é uniforme nem reprodutível**. Com isto queremos dizer que não podemos assumir nada com relação à taxa com que cada processo será executado, nem em relação a outros processos, nem em relação a diversas execuções de um mesmo processo. Uma das implicações disto é que os programas não podem ser feitos levando em consideração as temporizações de execução das instruções (p.ex.: não podemos fazer um programa que execute certo número de repetições de um “loop” e com

isso espere conseguir uma demora fixa de tempo, como um segundo). Isto ocorre porque não sabemos o momento em que a UCP será chaveada para outro processo, fazendo com que o atual tenha sua continuação retardada.

Para fixar melhor a diferença entre um programa e um processo, vejamos a seguinte comparação extravagante:

Suponha um padeiro não muito experiente. Para fazer um pão ele se utiliza de alguns ingredientes (farinha, sal, água, fermento, etc.) e segue a receita de um livro de receitas. Neste caso poderíamos estabelecer a seguinte comparação: O padeiro é a UCP, os ingredientes são as entradas e a receita é o programa (desde que o padeiro entenda tudo o que está escrito na receita, isto é, que a mesma esteja em uma **linguagem** apropriada). O **processo** neste caso corresponde ao ato de o padeiro estar executando o pão da receita com as entradas disponíveis.

Suponhamos agora que, enquanto o padeiro está amassando o pão (um dos passos indicados pela receita) ele é picado por uma abelha. Como é muito sensível, ele começa a chorar e julga extremamente importante cuidar da picada de abelha antes de acabar o pão. Para isto ele dispõe de novas entradas (os medicamentos) e de um novo programa (o livro de primeiros socorros). Terminado o curativo da picada o padeiro **volta a amassar o pão no mesmo ponto em que parou**. Aqui temos uma interrupção (a picada de abelha) fazendo a UCP (padeiro) chavear do processo de preparação do pão para o processo de curativo da picada. Outro fato importante aqui é que o processo de preparação do pão deve ser guardado com todo o seu estado corrente (isto é, em que ponto da receita ele está e qual a disposição de todos os componentes da massa e seu estado atual) para que possa ser reassumido **no mesmo ponto** em que foi interrompido.

Hierarquia de Processos

Em todos os S.O. de multiprogramação deve haver alguma forma de criar e terminar processos conforme o necessário à execução dos pedidos de usuários. Por isso um processo tem a possibilidade de gerar outros processos, que por sua vez podem gerar outros, e assim sucessivamente, gerando uma hierarquia de processos. Na geração de um novo processo, o processo gerado recebe o nome de **filho**, e o que pediu a geração recebe o nome de **pai**. Um pai pode possuir diversos filhos, mas cada filho só pode ter um pai.

Um exemplo de geração de processos é apresentado pela inicialização do UNIX. Quando o sistema é ligado, um processo chamado `init` é executado. Este processo lê um arquivo que diz quantos terminais existem; em seguida aguarda *login* nesses terminais. Quando um *login* é aceito, ele cria para esse terminal um *shell*; o *shell* por sua vez gera novos processos de acordo com os pedidos do usuário.

Estados dos processos

Durante a sua existência, os processos podem se apresentar, do ponto de vista do **sistema**, em diferentes estados. Apresentaremos os três mais importantes e os fatos que levam os processos a mudar de um estado a outro.

O primeiro estado a considerar consiste naquele em que um processo está efetivamente executando, isto é, está **rodando**.

Durante a sua existência, um processo pode necessitar interagir com outros processos. Por exemplo, um processo pode gerar uma saída que será utilizada por outro. Este é o caso dos processo `cat` e `grep` no exemplo abaixo:

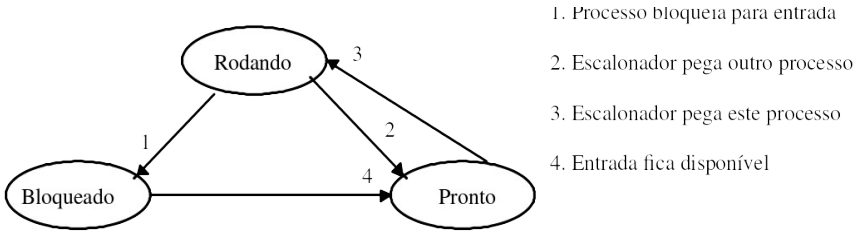
```
cat cap1 cap2 cap3 | grep arquivo
```

o processo `cat` é utilizado para con**cat**enar arquivos e apresentar a saída concatenada na saída padrão do sistema `stdout` (normalmente o terminal do usuário); no caso do exemplo, ele concatena os arquivos de nomes `cap1`, `cap2` e `cap3`, na ordem. O programa `grep` é utilizado para selecionar as linhas nas quais aparece uma dada cadeia de caracteres, utilizando como entrada a entrada padrão, `stdin` (normalmente o terminal do usuário); no nosso exemplo, procura as linhas em que aparece a palavra `arquivo`. Como a saída do processo `cat` está ligada à entrada do processo `grep` por meio de um *pipe*, a linha acima faz com que a saída na console seja todas as linhas que contém a palavra `arquivo` nos três arquivos citados (`cap1`, `cap2` e `cap3`).

No entanto, durante a execução desses dois processos, **como não sabemos a que taxa cada um será executado**, pode ocorrer de que o processo `grep` queira ler um dado de entrada que ainda não foi gerado pelo `cat`. Quando isto ocorre, o processo `grep` deve ser **bloqueado** até haver entradas.

Outro estado a considerar é quando o processo tem todas as condições de executar, mas não pode pois a UCP foi alocada para a execução de um outro processo. Neste caso dizemos que o processo está **pronto**.

Os três estados citados e suas inter-relações podem ser apresentados como na figura abaixo:



Onde as transições entre os estados estão numeradas de acordo com a notação:

1. bloqueio por falta de entrada
2. escalonador selecionou um outro processo
3. escalonador selecionou este processo
4. entrada ficou disponível

Em alguns sistemas, o processo deve requisitar o bloqueio, quando notar que não dispõe de entradas. Em outros sistemas (como o UNIX), o bloqueio é realizado automaticamente pelo próprio sistema, sem que o usuário precise se ocupar disso durante a programação.

O escalonador (*scheduler*) citado acima é uma parte do S.O. responsável pelo chaveamento da UCP entre os diversos processos, tendo como principal objetivo a conciliação da necessidade de eficiência do sistema como um todo e de justiça para com os processos individuais.

Implementação de processos

Para a implementação de processos, além dos programas a serem executados, devem ser guardadas algumas informações, necessárias ao escalonador para permitir que um processo seja reassumido exatamente no ponto em que foi interrompido. Esses dados são armazenados na chamada **tabela de processos**, que consiste em um vetor de estruturas com uma entrada por processo. Esta estrutura contém dados como: contador de programa (PC), ponteiro de pilha (SP), estado dos arquivos abertos, registradores, além de diversas outras informações.

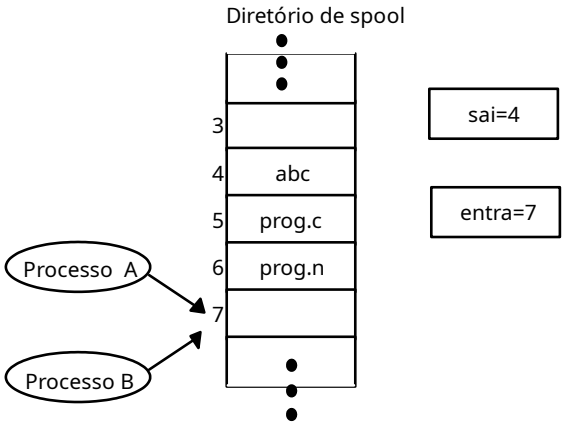
2.2. Comunicação entre processos

2.2.1. Condições de disputa

Quando existe compartilhamento, entre processos, de uma memória, na qual cada um pode escrever ou ler, podem ocorrer as chamadas condições de disputa. Exemplificaremos isto através de um *spooler* de impressora (que corresponde a um programa que permite diversos usuários utilizarem uma mesma impressora, simplesmente fazendo com que cada um envie o nome de um arquivo a ser impresso, e imprimindo os arquivos na ordem em que foram solicitados pelos diversos usuários).

Consideraremos que o programa funciona lendo os nomes dos arquivos a imprimir de um diretório de *spool*. Cada programa que deseja a impressão de um arquivo envia o nome do mesmo para o diretório de *spool*.

Consideraremos o diretório de *spool* como organizado em diversas unidades numeradas, sendo que cada unidade pode conter o nome de um arquivo (veja figura abaixo)



Temos também duas variáveis, `entra` e `sai` que são utilizadas respectivamente para indicar qual a próxima unidade de diretório vazia (e que portanto pode ser ocupada por um novo nome de arquivo) e qual a próxima unidade a ser impressa.

Suponhamos que, no caso indicado na figura acima, onde `entra=7`, um processo A resolve requisitar a impressão de um arquivo chamado `a.prg`. Para isto ele precisa ler o valor da variável `entra`, colocar o nome do arquivo na unidade correspondente, e incrementar o valor de `entra` de um. Suponhamos então, no nosso caso, que o processo A leu `entra=7`, após o que foi interrompido pelo escalonador, que determinou o início da execução do processo B. Este processo (B), resolve então imprimir o arquivo `b.prg`. Para isto lê o valor de `entra`, onde encontra 7, coloca `b.prg` na unidade 7 do diretório de *spool*, e incrementa `entra` para 8. Após isto o escalonador determina a volta da execução do processo A, que retorna exatamente no ponto onde estava. Então, para o processo A, o valor de `entra` continua sendo 7 e, portanto, ele colocará `a.prg` na unidade 7 do diretório de *spool* (exatamente onde tinha sido colocado `b.prg`) e incrementa o valor de `entra` que leu (isto é, 7) gerando como resultado 8, que será armazenado em `entra`. O resultado de tudo isto é que o pedido de impressão de `b.prg` desapareceu, e portanto o processo que o requisitou não será servido.

Isto é um exemplo do que chamamos de uma condição de disputa.

2.2.2. Seções Críticas

Claramente as condições de disputa, como a apresentada acima, devem ser evitadas, e os S.O. devem ser construídos de forma a evitar disputas entre os diversos processos.

Note que o problema de disputa ocorreu, no programa de *spool* da impressora, devido ao fato de termos **dois** processos acessando "simultaneamente" os dados compartilhados entre os mesmos. Isto nos indica que as condições de disputa podem ser evitadas proibindo que mais de um processo leia e escreva simultaneamente em uma área de dados compartilhada. Isto é o que se chama de **exclusão mútua**.

Durante a maior parte do tempo um processo executa computações internas, que não requerem acesso a dados de outros processos. Os trechos de programa em que os processos estão executando computações sobre dados compartilhados com outros processos são chamados de **seções críticas**. Para evitar disputas basta então garantir que não haverão dois processos simultaneamente em suas seções críticas.

Para haver uma cooperação eficiente e correta entre os processos, devemos satisfazer as seguintes condições:

1. não podem haver dois processos simultaneamente em suas seções críticas;
2. não são feitas suposições sobre a velocidade relativa dos processos e sobre o número de UCPs;
3. nenhum processo parado **fora** de sua região crítica pode bloquear outros processos; e
4. nenhum processo deve esperar um tempo arbitrariamente longo para entrar em sua região crítica.

2.2.3. Exclusão mútua com espera ocupada

Desabilitando interrupções

A forma mais simples de garantir a exclusão mútua é fazer com que cada processo desabilite interrupções ao entrar na região crítica, e as reabilite imediatamente antes de sair. Isto impede que a UCP seja chaveada para outro processo, pois o chaveamento é realizado através de uma interrupção periódica vinda de um relógio que ativa o escalonador.

Esta solução apresenta os seguintes problemas:

a. os usuários devem ter o direito de desabilitar interrupções, o que significa que se algum se esquecer de reabilitá-las, o S.O. não poderá mais executar
b. se o computador possuir várias UCPs o método não funciona, pois somente serão desabilitadas as interrupções da UCP que estiver rodando o programa

Disto concluímos que a desabilitação de interrupções, necessária a algumas tarefas do *kernel*, deve ser restrita ao mesmo.

Variáveis de Comporta

Podemos pensar numa solução em que se estabeleça uma variável auxiliar, denominada **variável de comporta** (*lock variable*), que quando em 0 indica que a região crítica está livre, e quando em 1 indica que a mesma está ocupada.

Desta forma, podemos fazer com que cada processo, antes de entrar, teste o valor da comporta: se for 0 (aberta), coloca em 1 e prossegue o processamento, colocando em 0 quando terminar, e se for 1 (fechada) aguarda até se tornar 0.

O grande problema com esta solução é que a disputa apenas se transferiu da região crítica para a variável de comporta (pense no que ocorre se um processo for interrompido imediatamente depois de ler o valor da variável e antes de alterá-lo...).

Alternância estrita

Esta é uma solução que obriga que a região crítica seja dada a um dos processos por vez, em uma estrita alternância. O algoritmo apresentado abaixo representa um exemplo disto, com a utilização de uma variável `vez`, que indica de qual processo é a vez de entrar na região crítica:

```
void proc_0() {
  while (true) {
    while (vez != 0) /* espera */ ;
    secao_critica0();
    vez = 1;
    secao_normal0();
  }
}

void proc_1() {
  while (true) {
    while (vez != 1) /* espera */ ;
    secao_critica1();
    vez = 0;
    secao_normal1();
  }
}
```

O teste contínuo de uma variável na espera de um certo valor é chamado de espera ocupada, e representa, como é evidente, um enorme desperdício de UCP.

O problema com a solução de alternância estrita é que requer que os dois processos se alternem precisamente, o que significa que o número de acessos de cada processo deve ser exatamente igual ao do outro. Além disto, existe uma violação da regra 3 apresentada acima, pois se um dos processos pára fora de sua seção crítica, o outro não poderá prosseguir normalmente, pois após entregar a seção crítica para o outro processo não pode mais pedi-la novamente.

Solução de Peterson

Por muitos anos a solução do problema da exclusão mútua não foi descoberta. O primeiro a propor uma solução funcional foi Dekker, sendo que entretanto esta solução era inaplicável na prática por ser extremamente complicada e dispendiosa de tempo de computação.

Em 1981, Peterson descobriu uma solução extremamente simples, que vai apresentada no programa seguinte:

```
#define N 2 /* numero de processos (precisa ser 2 no código abaixo) */

int vez;
int interessados[N];

void entra_regiao(int processo) {
  int outro = 1 - processo;
  interessados[processo] = true;
  vez = processo;
  while (vez == processo && interessado[outro]) /* espera */ ;
}

void deixa_regiao(int processo) {
 interessado[processo] = false;
}
```

A solução consiste em fazer com que um processo, antes de entrar na região crítica execute a rotina `entra_regiao` com o seu número. Ao terminar a utilização da região crítica o processo executa `deixa_regiao`.

É interessante estudar o funcionamento desta solução, para os diversos casos possíveis de interação entre dois processos que executam `entra_regiao` antes de entrar na região crítica e `deixa_regiao` ao sair da mesma. Como um exemplo, suponhamos que os dois processos executam as chamadas de `entra_regiao` praticamente ao mesmo tempo. Neste caso, ambos indicarão o seu interesse colocando a respectiva entrada do vetor `interessados` em `true`. Após isto, um dos processos deixará em `vez` o seu valor (pois alterará esta variável por último), e portanto habilitará o outro processo a continuar, aguardando até que o outro processo indique que não mais está interessado, executando `deixa_regiao`.

A instrução TSL

Nesta solução, pedimos auxílio do *hardware* do computador, que deve possuir uma instrução correspondente à aqui chamada `tsl` (de *Test and Set Lock*). Esta corresponde a uma instrução de máquina que lê o valor de uma variável de memória e armazena na mesma um valor diferente de 0, tudo em uma única instrução indivisível. Se por acaso o sistema tiver várias UCP, o próprio *hardware* deve prover arbitração de forma a que uma única UCP por vez tenha acesso a essa variável.

A utilização dessa instrução para resolver o problema da exclusão mútua pode ser apresentada pela codificação assembler fictícia apresentada abaixo:

```
entra_regiao:
   tsl reg, flag
   cmp reg, #0
   jnz entra_regiao
   ret
deixa_regiao:
   move flag, #0
   ret
```

onde consideramos que a variável de comporta se chama `flag`.

O problema tanto desta solução como da de Peterson, é que os próprios processos devem chamar `entra_regiao` e `sai_regiao`. Se um deles for mal programado ou trapaceiro pode monopolizar a utilização da região crítica.

2.2.4. Sleep e Wakeup

Tanto a solução de Peterson como a utilização da instrução TSL têm o grave inconveniente de desperdiçar tempo de UCP nos *loops* de espera para a entrada na região crítica. Além disto apresentam um outro problema quando tratamos de processos com prioridades diferentes. Suponha dois processos: um, chamado H, de alta prioridade e outro, chamado L de baixa prioridade. Se o processo L estiver executando em sua região crítica quando o processo H é selecionado para execução, então, se o processo H tenta entrar em sua região crítica não pode, pois o processo L está dentro da mesma e, portanto fica em um *loop* de espera. Mas, como H tem alta prioridade, o processo L não poderá executar até que H termine, pois apresenta prioridade mais baixa. Temos neste caso uma situação chamada de ***deadlock***, onde nenhum dos processos pode prosseguir pois está aguardando alguma condição que somente pode ser atingida pela execução do outro.

Para isto definimos duas rotinas `sleep` e `wakeup`, que realizam a espera através do **bloqueio** do processo, ao invés do desperdício do tempo de UCP.

`sleep` faz com que o processo que está executando seja transferido do estado de **rodando** para o de **bloqueado**. `wakeup` pega um processo em estado **bloqueado** e o transfere para o estado **pronto**, colocando-o disponível para execução quando o escalonador julgar adequado.

Para exemplificar a utilização de `sleep` e `wakeup`, vejamos o problema do produtor e do consumidor. Neste problema clássico existem dois processos: um, chamado **produtor**, que coloca dados em um *buffer*, e outro, chamado **consumidor**, que retira dados do *buffer*. O *buffer* apresenta uma capacidade finita de reter dados, de forma que surgem problemas em duas situações:

a. quando o produtor deseja colocar mais dados em um *buffer* cheio

b. quando o consumidor deseja retirar dados de um *buffer* vazio

Em ambos os casos, fazemos com que o processo que não pode acessar o *buffer* no momento execute um `sleep` e seja bloqueado, somente sendo acordado quando o outro processo alterar a situação do *buffer* e executar `wakeup`.

No programa abaixo temos uma tentativa de resolver o problema do produtor-consumidor com a utilização de `sleep` e `wakeup`. A variável `cont` é compartilhada entre os dois processos:

```
#define N 100

int cont = 0; /* buffer vazio */

void produtor() {
  while (true) {
    produz_item(&item);
    if (cont == N) sleep(); /* se cheio, dorme */
    entra_item(item);
    /* Se estava vazio, acorda o consumidor */
    if (++cont == 1) wakeup(consumidor);
  }
}

void consumidor() {
  while (true) {
    if (cont == 0) sleep(); /* se vazio, dorme */
    remove_item(&item);
    /* se estava cheio acorda o produtor */
    if (--cont == N-1) wakeup(produtor);
    consome_item(item);
  }
}
```

Infelizmente este programa apresenta um problema na utilização da variável `cont`. Veja nos seguintes passos:

1. consumidor lê `cont=0`;
2. escalonador interrompe consumidor e roda produtor;
3. produtor coloca item e incrementa `cont` para 1;
4. como descobre que `cont=1`, chama `wakeup` para o consumidor;
5. consumidor volta a executar, mas como já estava rodando o `wakeup` é perdido;
6. para ele `cont=0` (foi o valor lido) e portanto, dorme;
7. em algum momento o produtor lota o *buffer* e também dorme.

Temos portanto uma condição em que tanto o produtor como o consumidor estão dormindo, não podendo ser acordados pelo outro, o que caracteriza novamente um *deadlock*.

Para este caso simples, o problema pode ser resolvido com a inclusão de um bit de espera de `wakeup`. Assim, quando um `wakeup` for enviado para um processo já acordado, o bit será marcado, sendo que a próxima vez que o processo realizar `sleep`, se o bit estiver marcado então o processamento prosseguirá como se o `wakeup` tivesse sido automático.

Entretanto esta solução não resolve o problema geral. Pode-se, por exemplo, construir casos com três processos onde um bit de espera não é suficiente. O aumento para três ou mais bits poderia resolver esses casos, porém o problema geral ainda permaneceria.

2.2.5. Semáforos

Para resolver este problema, Dijkstra propôs em 1965 a utilização de uma variável inteira que conta o número de `wakeup` realizados. A essa variável se deu o nome de **semáforo**.

Para trabalhar com os semáforos Dijkstra propôs duas primitivas:

`P(s)`: generalização de `sleep`, que pode ser definida como:

```
void P(semaforo s) {
  if (s.cont > 0)
    s.cont = s.cont - 1;
  else {
    insere_na_fila(s.fila, processo_corrente);
    sleep();
  }
}
```

`V(s)`: generalização de `wakeup`, que pode ser definida como:

```
void V(semaforo s) {
  if (s.cont == 0 && !FilaVazia(s.fila)) {
    p = tira_da_fila(s.fila);
    wakeup(p);
  }
  else
    s.cont = s.cont + 1;
}
```

onde `s` é o semáforo, `s.cont` é a variável inteira associada ao semáforo `s`, `s.fila` é a fila associada ao mesmo semáforo (talvez algum tipo de `struct`), e supomos a existência de rotinas para lidar com as filas.

Veja que nas representações acima, consideramos todas as ações de P(s) e V(s) como **indivisíveis**, isto é, o processamento é realizado sem interrupções.

A solução do produtor-consumidor com semáforos é apresentada abaixo, onde utilizamos três variáveis semáforo:

`mutex` : semáforo binário que garante a exclusão mútua dentro da região crítica
`vazio` : que indica o número de posições vazias no *buffer*
`cheio` : que indica o número de posições cheias no *buffer*
Nota: `inicia()` deve ser chamada antes de `produtor()` e `consumidor()` serem chamados.

```
#define N 100 /* buffer tem 100 posições */

semaforo mutex, vazio, cheio;

void inicia() {
  mutex.cont=1;
  vazio.cont=N;
  cheio.cont=0;
}

void produtor() {
  int item;

  while (true) {
    produz_item(&item);
    P(vazio); /* menos um vazio */
    P(mutex); /* exclusão mútua */
    entra_item(item);
    V(mutex);
    V(cheio); /* um a mais cheio */
  }
}

void consumidor() {
  int item;

  while (true) {
    P(cheio); /* menos um cheio */
    P(mutex); /* exclusão mútua */
    remove_item(&item);
    V(mutex);
    V(vazio); /* um vazio a mais */
    consome_item(item);
  }
}
```

A melhor forma de ocultar uma interrupção é através da utilização de semáforos. Por exemplo, associamos um semáforo chamado `disco` à interrupção de disco, com o valor inicialmente de 0. Assim, quando um processo necessita de dados do disco executa um `P(disco)`, ficando bloqueado. Ao surgir a interrupção de disco esta executa `V(disco)`, desbloqueando o processo requisitante.

2.2.6. Contadores de Evento

Pode ser possível a solução do problema também sem a utilização da exclusão mútua. Para isto introduzimos o conceito de variável contadora de eventos, devido a Reed e Kanodia em 1979. Numa variável contadora de evento as únicas operações possíveis são (onde `E` é o contador de eventos):

`read(E)`: retorna o valor corrente de `E`
`advance(E)`: incrementa `E` de 1

`await(E, v)`: espera até que $E \geq v$

Os contadores de eventos são sempre incrementados, e nunca decrementados. A solução do produtor-consumidor com contadores de eventos é apresentada abaixo (supõe-se que os contadores de evento são automaticamente inicializados em zero):

```
#define N 100

contador_de_evento entrada, saida;

void produtor() {
  int item, sequencia = 0;

  while (true) {
    produz_item(&item);
    sequencia++;
    await(saida, sequencia - N);
    entra_item(item);
    advance(entrada);
  }
}

void consumidor() {
  int item, sequencia = 0;

  while (true) {
    sequencia++;
    await(entrada, sequencia);
    remove_item(&item);
    advance(saida);
    consome_item(item);
  }
}
```

2.2.7. Monitores

Os semáforos resolvem o problema de acesso às regiões críticas, entretanto apresentam grande dificuldade de utilização. Por exemplo, no código para o produtor-consumidor, se revertermos acidentalmente o `P(vazio)` com o `P(mutex)` no produtor, ocorrerá *deadlock* quando o *buffer* estiver cheio, pois o produtor já terá pedido a exclusão mútua (com a execução de `P(mutex)`) quando percebe que o *buffer* está cheio (com `P(vazio)`) e então pára. Mas então o consumidor não pode mais retirar ítens do *buffer* pois não poderá entrar na região crítica.

Isto mostra que a programação com semáforos deve ser muito cuidadosa. Para evitar essas dificuldades, Hoare (em 1971) e Brinch Hansen (em 1972) propuseram uma primitiva de sincronização de alto nível conhecida como monitor.

Em um monitor se agrupam conjuntamente rotinas, variáveis e estruturas de dados. Os monitores apresentam as seguintes características:

- os processos chamam rotinas no monitor quando desejam, mas não têm acesso a suas variáveis e estruturas de dados internas
- apenas um processo pode estar ativo no monitor (executando rotinas do monitor) em cada instante
- o monitor é reconhecido pelo compilador e, portanto, tratado de uma forma especial

- as primeiras instruções de cada rotina do monitor realizam um teste para verificar se existe outro processo ativo no monitor no mesmo instante. Se houver, o processo chamante é suspenso, e se não houver o processamento prossegue
- o compilador é o responsável pela implementação da exclusão mútua entre as rotinas do monitor (em geral através de um semáforo binário)
- quem escreve as rotinas do monitor não precisa saber como a exclusão mútua será implementada, bastando colocar todas as seções críticas em rotinas de monitor

Para possibilitar o bloqueio de processos quando estes não podem prosseguir introduziram-se as variáveis de condição, além das operações `wait` e `signal`. Quando uma rotina de monitor não pode continuar executa um `wait` em alguma variável de condição (por exemplo: `cheio`, quando o produtor encontra o *buffer* cheio). Um outro processo então pode acordá-lo executando um `signal` na mesma variável de condição. Para evitar que dois processos estejam ativos no monitor simultaneamente, devemos ter uma regra que diga o que deve ocorrer após `signal`. Hoare propôs que o processo recém acordado fosse executado. Brinch Hansen propôs que o processo que executa um `signal` deve deixar imediatamente o monitor (isto é, o `signal` deve ser a última operação realizada por uma rotina de monitor). Esta última é mais simples de implementar e conceitualmente mais fácil: se um `signal` é executado em uma variável de condição em que diversos processos estão esperando, apenas um deles (determinado pelo escalonador) é revivido.

A solução do problema do produtor-consumidor com a utilização de monitor é apresentado pelo pseuso-código abaixo.

```
monitor ProdutorConsumidor {
  condition cheio, vazio;
  int cont = 0;

  void coloca(int item) {
    if (cont == N) wait(cheio);
    entra_item(item);
    if (++cont == 1) signal(vazio);
  }

  void retira(int *pitem)
    if (cont == 0) wait(vazio);
    remove_item(pitem);
    if (--cont = N - 1) signal(cheio);
  }
};

void produtor() {
  while (true) {
    int item = produz_item();
    ProdutorConsumidor.coloca(item);
  }
}

void consumidor() {
  while (true) {
    int item;
    ProdutorConsumidor.retira(&item);
    consome_item(item);
  }
}
```

As operações `wait` e `signal` são muito parecidas com as de `sleep` e `wakeup`, que tinham problemas graves de disputa. No caso atual a disputa não ocorre, pois no monitor não podem haver duas rotinas simultaneamente.

Os monitores possibilitam uma grande facilidade para a programação paralela. Entretanto, possuem alguns inconvenientes:

a. são uma construção de alto nível, que deve ser reconhecida pelo compilador, isto é, a linguagem que os implementa deve reconhecer os comandos de monitor empregados (muito poucas linguagens de fato os implementam); e
b. eles se baseiam (tanto quanto os semáforos) na consideração da existência de uma memória comum compartilhada por todos os processadores. Portanto são inaplicáveis para sistemas distribuídos, onde cada processador conta com sua própria memória independente.

2.2.8. Passagem de Mensagens

Para possibilitar o trabalho com sistemas distribuídos, e também aplicável a sistemas de memória compartilhada, temos o conceito de **passagem de mensagens**. Este método usa as seguinte primitivas:

`send`: envia uma mensagem para um destino dado:

```
send(destino,&mensagem);
```

`receive`: recebe uma mensagem de uma origem especificada

```
receive(origem,&mensagem);
```

Se nenhuma mensagem estiver disponível no momento de executar `receive`, o receptor pode bloquear até que uma chegue.

Se pensamos na conexão de processadores por uma rede, pode ocorrer de uma mensagem ser perdida. Para se resguardar disto o processo que envia e o que recebe podem concordar em que, tão logo uma mensagem seja captada pelo receptor, este envie uma outra mensagem acusando a recepção. Assim, se o processo que envia não receber a recepção dentro de certo tempo, pode concluir que a mensagem foi perdida e tenta a transmissão novamente. Estas mensagens de volta são mensagens especiais chamadas de mensagens de **reconhecimento**.

Suponha agora que a mensagem original é recebida corretamente, mas o reconhecimento é perdido. Neste caso, o transmissor irá retransmitir a mensagem, e o receptor irá recebê-la duas vezes. Portanto, deve ser possível a distinção entre uma mensagem antiga retransmitida e uma nova mensagem, o que é normalmente realizado pela colocação de números consecutivos em cada mensagem original.

Sistemas de mensagens também precisam saber exatamente quais os nomes dos diversos processos, de forma a identificar corretamente a origem e o destino de uma mensagem. Normalmente isto é conseguido com uma nomeação do tipo **processo@máquina**. Se o número de máquinas for muito grande e não houver uma autoridade central, então pode ocorrer que duas máquinas se dêem o mesmo nome. Os problemas de conflito podem ser grandemente diminuídos pelo agrupamento de máquinas em **domínios**, sendo o endereço formado por **processo@máquina.domínio**.

Outro fator importante a considerar é o de **autenticação**, de forma a garantir que os dados estão sendo realmente recebidos do servidor requisitado, e não de um impostor, e também para um servidor saber com certeza qual o cliente que requisitou. Em geral a encriptação das mensagens com alguma chave é útil.

Utilizando a passagem de mensagens podemos resolver o problema do produtor-consumidor como apresentado no programa abaixo:

```
#define N 100 /* número de slots no buffer */

void produtor() {
  int item;
  message m;

  while (true) {
    produz_item(&item);
    /* Aguarda mensagem vazia */
    receive(consumidor, &m);
    /* constroi mensagem */
    faz_mensagem(&m, item);
    /* envia ao consumidor */
    send(consumidor, &m);
  }
}

void consumidor() {
  int item, i;
  message m;

  /* envia N mensagens vazias */
  for (i = 0; i < N; ++i) send(produtor, &m);
  while (true) {
    receive(produtor, &m);
    extrai_item(&m, &item);
    consome_item(item);
    /* envia resposta vazia */
    send(produtor, &m);
  }
}
```

Nesta solução assumimos que todas as mensagens têm o mesmo tamanho e que mensagens enviadas mas ainda não recebidas são automaticamente armazenadas pelo sistema operacional.

No início são enviados N vazios para permitir ao produtor encher todo o *buffer* antes de bloquear. Caso o *buffer* esteja cheio, o produtor bloqueia no `receive` e aguarda a chegada de uma nova mensagem vazia.

A passagem de mensagens pode ser feita de diversas formas, das quais citaremos três:

1. assinalando-se um endereço único para cada processo e fazendo as mensagens serem endereçadas aos processos;
2. com a utilização de ***mailbox***, uma nova estrutura de dados que representa um local onde diversas mensagens podem ser *bufferizadas*. Neste caso, os endereços em `send` e `receive` são as *mailbox* e não os processos. Se um processo tenta enviar para uma *mailbox* cheia ou receber de uma *mailbox* vazia ele é automaticamente suspenso até que a *mailbox* apresente as condições para a operação;
3. eliminando toda a bufferização, fazendo com que, se o processo que envia executa `send` antes do de recepção então ele é bloqueado até que este último esteja pronto (e vice-versa). Este mecanismo é conhecido como ***rendezvous.***

Em UNIX, a comunicação entre processos é realizada por passagem de mensagens através de ***pipes***, que no fundo são como *mailboxes*, com a diferença de que o número de mensagens transmitidas por um *pipe* não precisa ser igual ao de mensagens recebidas, desde que o total das mensagens (todos os seus bytes) seja lido. Por exemplo, se um processo envia 10 mensagens de 100 bytes cada e outro lê uma mensagem de 1000 bytes, então não há problema (no caso de *mailboxes*, deveria haver uma correspondência completa entre mensagens transmitidas e lidas).

2.2.9. Equivalência de primitivas

Além das primitivas apresentadas, diversas outras foram propostas, sendo a lista total muito extensa.

No entanto, algumas destas primitivas são essencialmente equivalentes (pelo menos enquanto se considera uma única UCP). Vejamos agora as equivalências entre semáforos, monitores e passagem de mensagem.

Utilizando semáforos para implementar Monitores e Passagem de Mensagens

Se o S.O. fornece semáforos, então o projetista de um compilador pode incluir monitores em sua linguagem como segue:

a. Associado a cada monitor existe um semáforo binário, `mutex`, inicialmente 1, que controla a entrada no monitor, e um semáforo adicional, inicialmente 0, para cada variável de condição
b. Quando uma rotina de monitor é chamada, a primeira coisa que executa é um `P(mutex)` do correspondente monitor. Se o monitor já estiver em uso, o processo será bloqueado
c. Quando sai do monitor, a rotina faz um `V(mutex)`, permitindo a outros processos utilizarem o monitor
d. `wait` em uma variável de condição `c` é compilado numa seqüência de três operações em semáforos: `V(mutex); P(c); P(mutex);`
e. `signal` em uma condição `c` é implementado por um `V(c)`

É interessante estudar a operação deste processo para o caso do produtor-consumidor. O semáforo `mutex` garante que cada processo tem acesso exclusivo ao monitor em sua seção crítica. Vejamos os seguinte processo:

1. o consumidor começa antes e verifica que não tem dados para prosseguir
2. realiza um `wait` em `vazio`, o que causa um `V(mutex)` e um `P(vazio)`, sendo nesse instante bloqueado
3. o produtor descobre que `cont=1` e realiza um `signal(vazio)`, para acordar o consumidor. Neste instante tanto o produtor como o consumidor estão ativos no monitor simultaneamente. Entretanto isto não causa dano pois convencionamos que o `signal` é a última operação a ser executada no monitor por um processo
4. Se o produtor faz `V(mutex)` e deixa o monitor antes do consumidor ter feito `P(mutex)`, então é possível que o produtor entre novamente antes que o consumidor possa executar seu `P(mutex)`. No entanto não temos nenhum dano, pois o consumidor não pode realizar nenhuma inspeção ou alteração em qualquer variável compartilhada antes do `P(mutex)`. Se o produtor estiver no monitor neste momento, então `mutex` será zero e o consumidor será bloqueado até o produtor deixar o monitor.

Na implementação de **passagem de mensagens** (variação com *mailbox*) com a utilização de semáforos procedemos da seguinte forma:

i. para cada processo associamos um semáforo, de valor inicial zero, que será utilizado para o bloqueio do processo;
ii. criamos, em uma área compartilhada pelos processos os *mailboxes*, cada um consistindo de *slots* em uma lista ligada;
iii. a cada *mailbox* associamos duas variáveis, indicando quantos *slots* estão cheios e quantos vazios;
iv. a cada *mailbox* associamos duas filas: uma de processos bloqueados por não poder enviar para esse *mailbox* e outra de processos bloqueados por não poder receber dele;
v. protegemos a área compartilhada com um semáforo `mutex`, que será utilizado para garantir a exclusão mútua;

As primitivas agirão então da seguinte forma:

a. `send` para *mailbox* com algum *slot* vazio ou `receive` de mailbox com *slot* cheio: executa `P(mutex)`; manipula a mensagem; atualiza contadores; atualiza elementos de ligação das listas; executa `V(mutex)` e retorna;
b. `receive` em *mailbox* vazio: executa `P(mutex)`; o processo se coloca na fila de **recepção** do *mailbox* e executa a seqüência: `V(mutex); P(sp); P(mutex);` (onde chamamos de `sp` o semáforo correspondente ao processo que executou o `receive`);
c. `send` em *mailbox* cheio: o mesmo que para `receive` em *mailbox* vazio, só que o processo se coloca na fila de **transmissão**;
d. considerando a possibilidade de existência de processos esperando nas filas, cada vez que um `receive` for completado, deve testar a fila de transmissão para ver se é necessário acordar algum processo. O mesmo deve fazer o `send`, testando a fila de **recepção**.

Utilizando monitores para implementar semáforos e mensagens

Para a implementação de semáforos, prosseguimos da seguinte forma:

i. estabelecemos um contador e uma lista ligada para cada semáforo;
ii. estabelecemos uma variável de condição para cada processo.

Com essas estruturas, realizamos a implementação das funções `P` e `V` da seguinte forma:

`P(s)`**:** entra no monitor; verifica se o contador associado com o semáforo `s` é maior que zero e, se for, decrementa o contador e sai do monitor, se não for, coloca o número do processo na lista ligada daquele semáforo, executando a seguir um `wait` na variável de condição correspondente; sai do monitor.

`V(s)`: entra no monitor; se houverem entradas na lista ligada de `s`, remove uma e faz um `signal` na variável de condição do processo, senão, incrementa contador associado; sai do monitor.

Podemos implementar mensagens (variação de *mailbox*) de forma semelhante à que foi feita com semáforos, exceto que ao invés de estabelecermos um semáforo para cada processo estabelecemos uma variável de condição por processo.

Utilizando mensagens para implementar semáforos e monitores

Se dispomos de mensagens no S.O., os semáforos e monitores podem ser implementados facilmente com a utilização de um truque, que é a construção de um processo especial, chamado **processo de sincronização**.

Para a implementação de semáforos fazemos:

i. o processo de sincronização mantém um contador e uma lista ligada para cada semáforo;
ii. `P` e `V` chamam rotinas de biblioteca (digamos `BP` e `BV`), que enviam mensagens ao processo de sincronização especificando qual a operação a ser executada e qual o semáforo a utilizar;
iii. após enviar mensagem ao processo de sincronização, as rotinas `BP` e `BV` fazem um `receive` para pegar a resposta do processo de sincronização;
iv. o processo de sincronização checa o contador para verificar se a operação requisitada pode ser completada. `V` sempre completa, mas `P` pode bloquear se o contador correspondente ao semáforo for igual a zero;
v. o processo de sincronização envia uma mensagem vazia de volta se a operação pode ser completada, o que faz com que o processo que requisitou a operação (que está, via rotina de biblioteca, em `receive`) seja desbloqueado;
vi. se a operação requisitada for um `P` em semáforo com contador igual a zero, o processo de sincronização coloca o identificador do processo que requisitou a operação na lista e não envia uma resposta, fazendo com que o processo fique bloqueado no `receive` da rotina `BP`;

vii. quando um V for feito em um semáforo com processos bloqueados, o processo de sincronização pega um desses processos e envia ao mesmo a mensagem resposta (fazendo com que o `receive` seja desbloqueado);
viii. O fato do processo de sincronização realizar apenas uma requisição por vez, impede disputas nos acessos aos semáforos.

Para implementar monitores, podemos implementar os semáforos e então implementar o monitor a partir dos semáforos. Existem também outras possibilidades.

2.3. Problemas clássicos de comunicação interprocessos

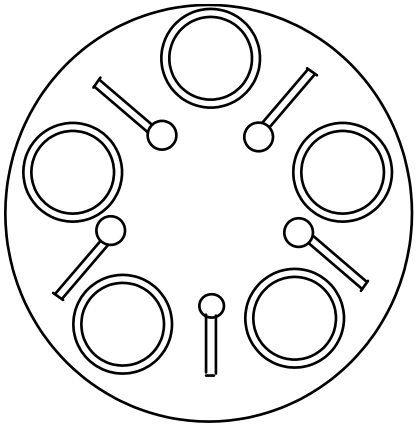
2.3.1. O problema do jantar dos filósofos (Dijkstra, 1965)

Este é um interessante problema que modela, de forma simplificada, a existência de diversos processos competindo por recursos em quantidade limitada.

O problema pode ser enunciado da seguinte forma:

a. Todos os 5 professores do Departamento de Filosofia estão participando de um jantar comum, sentados em torno de uma mesa redonda;
b. Em frente a cada filósofo existe um prato e, intercalado entre cada dois pratos existe um garfo;
c. Está sendo servida uma variação excessivamente escorregadia de *spaghetti*, de forma que, para comê-lo, cada filósofo precisa pegar os dois garfos que estão ao lado de seu prato;
d. cada filósofo alternadamente pensa e come;
e. quando sente fome, o filósofo tenta pegar os dois garfos ao seu lado;
f. se consegue pegar os garfos, come por alguns instantes e então devolve os garfos à mesa e volta a pensar.

Para maior clareza, veja a figura abaixo:



O problema consiste em escrever um programa para cada filósofo, de forma a que cada um faça o esperado (isto é, alterne períodos de pensamento e alimentação) e nenhum fique “encalhado” em alguma condição.

Como uma primeira tentativa, podemos sugerir o seguinte programa:

```
#define N 5 /* número de filosofos */

void filosofo(int i) {
  while (true) {
    pense();
    pegue_garfo(i);
    pegue_garfo((i + 1) % N);
    coma();
    solte_garfo(i);
    solte_garfo((i + 1) % N);
  }
}
```

Como pode ser visto facilmente, esta tentativa acarretará impasse (*deadlock*). Analise, por exemplo, o que acontece se todos pegam o garfo da esquerda (o garfo numero `i`) simultaneamente.

Para tentar resolver essa situação, podemos fazer com que o filósofo, após pegar o garfo da esquerda teste se o garfo da direita está disponível. Caso esteja pega, mas caso não esteja devolve o garfo da esquerda, aguarda um certo tempo, e tenta novamente.

Esta tentativa de solução leva a uma outra situação indesejada, chamada de fome (*starving*). Veja o que ocorre caso todos peguem o garfo da esquerda simultaneamente:

- testa se o garfo da direita está disponível;
- como não está, devolve garfo da esquerda e aguarda;
- pega novamente garfo da esquerda (momento em que, novamente, todos os filósofos pegam);
- novamente testa o garfo da direita, verifica que está ocupado e permanece indefinidamente neste ciclo.

Por definição, dizemos que chegamos a uma situação de fome quando todos os processos prosseguem fazendo algo (isto é, não estão bloqueados) mas não conseguem fazer progressos.

Pode-se tentar resolver esse problema fazendo com que cada filósofo aguarde um tempo aleatório antes de tentar novamente pegar os garfos. Esta idéia torna extremamente improvável a ocorrência de fome, mas, por outro lado, não a exclui totalmente, pois pode ainda ocorrer o caso em que sejam gerados números aleatórios iguais para todos os filósofos.

Podemos tentar resolver o problema através da utilização de semáforos:

- antes de pegar qualquer garfo, o filósofo executa `P(mutex)`
- após devolver os garfos, o filósofo executa `V(mutex)`.

Esta solução realmente impede que ocorra impasse ou fome, mas, por outro lado, somente permite que um único filósofo esteja comendo em cada instante. Como dispomos de 5 garfos, deve ser possível encontrar uma solução em que dois filósofos possam estar comendo simultaneamente.

A solução é apresentada abaixo (`inicia()` deve ser executada antes do inicio dos `filosofo(i)`):

```
#define N 5

typedef enum { PENSANDO, FAMINTO, COMENDO } Estados;

Estados estado[N];
semaforo mutex;
semaforo s[N];

int esquerda(int i) { return ((i - 1) + N) % N; }
int direita(int i)  { return (i + 1) % N; }

void inicia() {
  int i;
  mutex.cont = 1;
  for (i = 0; i < N; ++i) {
    estado[i] = PENSANDO;
    s[i].cont = 0;
  }
}

void pegue_garfo(int i) {
  P(mutex);
  estado[i] = FAMINTO;
  teste(i);
  V(mutex);
  P(s[i]);
}

void teste(int i) {
  if (estado[i] == FAMINTO &&
      estado[esquerda(i)] != COMENDO &&
      estado[direita(i)] != COMENDO) {
    estado[i] = COMENDO;
    V(s[i]);
  }
}

void solte_garfo(int i) {
  P(mutex);
  estado[i] = PENSANDO;
  teste(esquerda(i));
  teste(direita(i));
  V(mutex);
}

void filosofo(int i) {
  while (true) {
    pense();
    pegue_garfo(i);
    coma();
    solte_garfo(i);
  }
}
```

2.3.2. O problema dos leitores e escritores (Courtois, 1971)

Este problema realiza um modelamento da existência de diversos processos acessando uma base da dados comum.

Os processos são divididos em duas categorias: processos leitores e processos escritores. As regras podem ser definidas da seguinte forma:

- se diversos leitores querem acesso à base de dados, todos terão permissão
- somente um escritor pode ter acesso à base em cada instante, e durante esse tempo, nenhum leitor pode acessar a mesma.

Uma solução para esse problema com a utilização de semáforos é apresentada adiante (`inicia()` deve ser chamada antes dos outros processos):

```
semaforo mutex, db; /* controle da base */
int rc = 0;         /* numero de leitores */

void inicia() {
  mutex.cont = db.cont = 1;
}

void leitor() {
  while (true) {
    P(mutex);
    if (++rc == 1) P(db);
    V(mutex);
    le_base();
    P(mutex);
    if (--rc == 0) V(db);
    V(mutex);
    use_dados();
  }
}

void escritor() {
  while (true) {
    produz_dados();
    P(db);
    escreve_base();
    V(db);
  }
}
```

Nesta solução está embutido que os leitores têm prioridade sobre os escritores. Pode-se conseguir também uma solução em que os escritores tenham prioridade.

2.4. Escalonamento de processos

Chamamos de escalonamento (*scheduling*) de processos o ato de realizar o chaveamento dos processos ativos, de acordo com regras bem estabelecidas, de forma a que todos os processos tenham a sua chance de utilizar a UCP.

O escalonador (*scheduler*) é, portanto, a parte do S.O. encarregada de decidir, entre todos os processos prontos para executar, qual o que será rodado em cada instante.

Chamamos de **algoritmo de escalonamento** o método utilizado pelo escalonador para decidir, a cada instante, qual o próximo processo a ser executado.

Podemos estabelecer os seguintes critérios para um algoritmo de escalonamento:

a. **justiça**: cada processo deve conseguir sua parte justa do tempo de UCP;

b. **eficiência**: garantir uma ocupação de 100% do tempo da UCP;

c. **tempo de resposta**: minimizar o tempo de resposta a comandos de usuários interativos;

d. **minimização** do tempo que os usuários de lotes (*batch*) devem aguardar até conseguir a saída de seus pedidos;

e. **maximização** do número de serviços (*jobs*) processados por hora.

Como pode ser facilmente notado, vários critérios acima entram em contradição entre si, de forma que devemos prejudicar alguns para conseguir melhoras em outros. O escalonador deve, portanto, realizar algum compromisso entre os diversos critérios.

Para dificultar que os objetivos sejam alcançados, ainda temos o fato de que cada processo é único e imprevisível.

Com o objetivo de prevenir que um processo rode durante um tempo excessivo, a maioria dos computadores dispõe de um **relógio**, que é um dispositivo de temporização que interrompe a UCP a cada intervalo dado de tempo. A cada interrupção, o S.O. interrompe a execução do processo atual e roda o escalonador para decidir qual o próximo processo a ser executado.

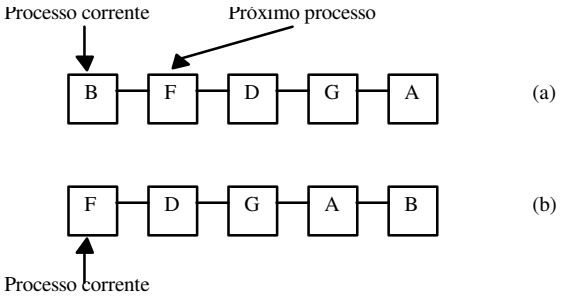
Esta estratégia de interromper um processo em execução é chamada de escalonamento **pre-emptivo** (***preemptive*** *scheduling*), e contrasta com as estratégias de execução até o término e escalonamento cooperativo.

Vejamos agora algumas formas de implementar o escalonamento.

2.4.1. Escalonamento Round-Robin

Neste método, a cada processo se dá um intervalo de tempo, chamado de ***quantum***, no qual se permite que ele execute. Se o processo ainda está rodando ao fim de seu *quantum*, a UCP é tomada deste processo e o escalonador selecionará um novo para rodar. A UCP também pode ser entregue a outro processo caso o processo bloqueie ou termine antes do fim do *quantum*. Neste caso, a UCP é entregue a outro processo no momento do término ou do bloqueio.

Para a implementação deste método, o escalonador mantém uma lista de processos executáveis (isto é, processos **prontos**). Quando um *quantum* termina sem o processo acabar, o mesmo é colocado no fim dessa lista. O escalonador seleciona, então, o primeiro processo dessa lista para execução. Veja a figura abaixo para melhor esclarecimento.



O valor do *quantum* deve ser cuidadosamente escolhido, pois, como existe um tempo para que o processo em execução seja interrompido, tenha seu estado atual salvo, o escalonador decida qual o próximo processo e o estado deste novo processo seja restaurado, se o valor do *quantum* for muito pequeno, teremos uma grande proporção do tempo de execução da UCP gasta com o processo de

escalonamento, que não é de interesse para os usuários. Por exemplo, se todo o chaveamento de processos leva 5ms, e o *quantum* for de 20ms, teremos uma utilização de 20% do tempo total para o chaveamento.

Por outro lado, se o valor do *quantum* for muito grande, isto pode acarretar um tempo de resposta muito grande para usuários interativos. Veja por exemplo se, para um chaveamento de 5ms escolhemos um *quantum* de 500ms. Neste caso o tempo gasto com o chaveamento é menos que 1% do tempo total. Entretanto, se temos 10 processos executando, o tempo de resposta para um processo pode ser de até 5s!

Devemos pois escolher um valor de compromisso, nem muito grande nem muito pequeno, de acordo com o tempo de chaveamento.

2.4.2. Escalonamento com prioridade

O esquema de *round-robin* trata igualmente todos os processos do sistema, sejam eles altamente urgentes e importantes, sejam eles de pouca importância ou sem urgência. Para poder oferecer um tratamento distinto a processos diversos, devemos introduzir o conceito de **prioridade**.

No instante da criação de um processo, associamos ao mesmo uma prioridade. Quando o escalonador tiver que escolher, entre os processos prontos, qual o que será executado em seguida, ele escolherá o de mais alta prioridade. Como esse esquema pode fazer com que um processo seja executado indefinidamente, não deixando espaço a outros processos, o escalonador pode, a cada vez que o processo é escalado para execução, decrementar sua prioridade. No momento em que sua prioridade fica abaixo da de um outro processo pronto ele é interrompido e o outro processo (agora com prioridade mais alta) é executado.

As prioridades podem ser associadas a processos de forma **estática** ou **dinâmica**. Dizemos que uma prioridade é **estática** quando ela é associada no momento da criação do processo (mesmo que depois ela seja decrementada para impedir o processo de monopolizar a UCP). Uma prioridade é **dinâmica** quando o escalonador é que decide seu valor, baseado em estatísticas sobre a execução deste processo em *quanta* anteriores.

Como exemplo de prioridades estáticas, podemos imaginar que em um sistema implantado em um quartel, os processos iniciados por um general devem ter prioridade sobre os processos iniciados por um coronel, e assim por diante, por toda a hierarquia.

Como exemplo de prioridade dinâmica, podemos citar um sistema que, quando percebe que um dado processo efetua muitas operações de entrada/saída, aumenta a sua prioridade. Isto é feito pois as operações de entrada/saída são realizadas independentemente da UCP, de forma que, se um processo executa muitas dessas operações, é conveniente deixar que ele requisite uma nova operação o mais rápido possível. Lembre-se que, ao requisitar uma operação de entrada/saída o processo faz uma chamada para o S.O. e fica bloqueado, aguardando o fim da operação. Assim, processos com grande utilização de E/S não significam muita carga para a UCP, pois enquanto suas transferências estão se executando a UCP pode estar alocada a outro processo que executa cálculos. Para implementar essa forma de prioridade, o escalonador calcula a fração de tempo que o processo utilizou do seu *quantum* na última chamada. Se o processo utilizou apenas uma pequena fração (e não terminou) é provavelmente porque está esperando dados de um dispositivo de E/S. Desta forma, podemos utilizar para prioridade, um valor proporcional ao inverso da fração de *quantum* ocupado.

Muitas vezes é útil agrupar os processos em classes de prioridades, fazendo um escalonamento em dois níveis:

i. escalonamento de prioridade entre as diversas classes, executando todos os processos das classes de prioridades mais altas antes das classes mais baixas;
ii. escalonamento *round-robin* dentro de cada classe, fazendo com que todos os processos de uma mesma classe revezem-se como explicado para o escalonamento *round-robin*.

2.4.3. Filas múltiplas

Sugerido inicialmente para o sistema CTSS. O fato importante aqui é que o computador forçava um chaveamento de processos extremamente lento, pois apenas um processo podia estar presente na memória em cada instante, fazendo com que os processos devessem ser lidos do disco e escritos no disco a cada chaveamento.

A utilização prática demonstrou que era necessário que os processos com muita utilização de UCP (também chamados de processos limitados pela UCP, *CPU-bound*) ganhassem um *quantum* grande de uma só vez, de forma a reduzir o número de chaveamentos executados nesse processo. Por outro lado, como já vimos, não podemos atribuir *quanta* grandes para todos os processos, pois isso implicaria tempo de resposta muito grande.

A solução encontrada foi o estabelecimento de classes de prioridade. Os processos de classes mais altas são escolhidos para execução mais freqüentemente que os de classes mais baixas. Por outro lado, os processos de classes mais baixas recebem um número maior de *quanta* para processamento. Assim, os processos da classe mais alta existente recebem 1 *quantum*, os processos da classe imediatamente abaixo recebem 2 *quanta*, os processos da classe seguinte, 4 *quanta*, e assim sucessivamente, nas potências de 2.

Adicionalmente a isso, cada vez que um processo utiliza todos os *quanta* que recebeu ele é abaixado de uma classe (fazendo com que seja escolhido menos freqüentemente, mas execute durante mais tempo).

Como exemplo, num processo que necessita de 100 *quanta* para executar, teremos execuções de 1, 2, 4, 8, 16, 32, 37. Isto é, ele é selecionado para execução 7 vezes (7 chaveamentos para este processo), sendo que no método de *round-robin* tradicional ele seria selecionado 100 vezes até acabar (100 chaveamentos).

Por outro lado, este método, como definido até aqui, tem o grave inconveniente de que, se um processo começa com uma grande quantidade de cálculos, mas em seguida se torna interativo (isto é, exige comunicação com o usuário), teríamos um tempo de resposta muito ruim. Para eliminar este problema, foi implementado o seguinte método: cada vez que um <RET> é teclado no terminal associado a um processo, este processo é transferido para a classe de prioridade mais alta, na esperança de que o mesmo irá se tornar interativo.

Esta solução funcionou bem até que um usuário descobriu que, durante a execução de um processo grande, bastava sentar no terminal e teclar aleatoriamente <RET> para ter o tempo de execução melhorado. Em seguida ele contou sua “descoberta” a todos. Vê-se então que a implementação prática de uma estratégia eficiente é muito mais difícil na prática do que em princípio.

2.4.4. Menor serviço (job) primeiro

As estratégias apresentadas anteriormente são úteis para sistemas de *time-sharing*. A estratégia de menor serviço primeiro (*shortest job first*) é útil fundamentalmente para sistemas de lotes (*batch*). Nestes sistemas, muitas vezes os usuários já têm uma boa estimativa dos tempos de execução do programa (em geral porque o programa é rodado constantemente).

A estratégia consiste em escolher para execução, entre todos os *jobs* disponíveis, aquele de menor tempo de execução.

Para ver como esta estratégia consegue melhorar o tempo médio de resposta do sistema, consideremos o seguinte caso: existem quatro *jobs* disponíveis (A, B, C e D), sendo um de 8 minutos (o A) e três de 4 minutos (B, C e D). Se o *job* de 8 minutos for escolhido antes, teremos a seguinte configuração:

tempo de resposta de A: 8 minutos

tempo de resposta de B: 8 + 4 = 12 minutos
tempo de resposta de C: 8 + 4 + 4 = 16 minutos
tempo de resposta de D: 8 + 4 + 4 + 4 = 20 minutos
tempo de resposta médio: 14 minutos.
Se escolhemos os três *jobs* de 4 minutos antes do de 8, teremos:
tempo de resposta de B: 4 minutos
tempo de resposta de C: 4 + 4 = 8 minutos
tempo de resposta de D: 4 + 4 + 4 = 12 minutos
tempo de resposta de A: 4 + 4 + 4 + 8 = 20 minutos
tempo de resposta médio: 11 minutos.

Esta estratégia garante o mínimo tempo de resposta médio possível, desde que se conheçam todos os serviços simultaneamente. Se os serviços não são todos conhecidos simultaneamente o tempo de resposta médio pode não ser o mínimo, mas ainda é uma boa aproximação.

2.4.5. Escalonamento dirigido a política

Outra forma de cuidar do escalonamento é fazer uma promessa ao usuário e procurar cumprí-la.

Um exemplo de uma promessa simples e fácil de cumprir é a de que, se existem **n** usuários ativos, então, cada um receberá aproximadamente 1/n do tempo de UCP. Para cumprir essa promessa podemos usar o seguinte método:

i. para cada usuário, é mantido um valor do tempo de UCP que ele já utilizou desde que entrou, que será chamado **tempo utilizado**;
ii. computa-se quanto tempo já se passou desde que o usuário iniciou sua seção e divide-se pelo número de usuários, gerando o que será chamado de **tempo destinado** ao processo;
iii. computa-se, para cada usuário, a **razão** entre o tempo utilizado e o tempo destinado;
iv. escolhe-se para execução aquele usuário que apresenta a menor razão.

2.4.6. Escalonamento em dois níveis

Até aqui, consideramos que todos os processos executáveis estão na memória. Este geralmente não é o caso, pois dificilmente a memória principal comportará todos os dados necessários. Precisaremos, pois, manter parte dos processos em disco.

O problema que surge é que o tempo para ativar um processo que está em disco é muito maior que o tempo necessário para ativar um processo que está na memória (uma a duas ordens de grandeza maior).

A melhor solução para isto é a utilização de um **escalonador em dois níveis**. O funcionamento será então da seguinte forma:

i. um subconjunto dos processos executáveis é mantido na memória;
ii. um outro subconjunto é mantido no disco;
iii. um escalonador (chamado **de baixo nível**) é utilizado para realizar o chaveamento (por qualquer dos métodos já descritos, ou outros semelhantes) apenas entre os processos que estão na memória;
iv. um outro escalonador (chamado **de alto nível**) é utilizado para trocar periodicamente o conjunto de processo que estão na memória (substituindo-os por alguns que estavam no disco).

Para realizar a escolha de qual conjunto de processos deve ser colocado na memória principal o escalonador de alto nível deve responder perguntas tais como:

- Quanto tempo se passou desde que o processo foi posto ou retirado?

- Quanto tempo de UCP o processo teve recentemente?
- Qual o tamanho do processo?
- Qual a prioridade do processo?

Exercícios

2.1. O que é uma condição de disputa?
2.2. O que acontece com o algoritmo apresentado para alternância estrita no caso de os dois processos estarem executando em UCP distintas que compartilham a mesma memória?
2.3. Se um computador não possui uma instrução do tipo TSL, mas possui uma que permite a troca entre os conteúdos de uma posição de memória especificada e o valor de certo registrador (de uma forma indivisível), pode essa instrução ser utilizada para implementar `entra_regiao` e `sai_regiao` (similares às do texto)? Caso seja possível, escreva o código.
2.4. De um esboço de como um S.O. que pode desabilitar interrupções poderá implementar semáforos. (Considere um sistema com uma única CPU).
2.5. Se você dispõe apenas de semáforos binários, mostre de que forma você poderia implementar semáforos arbitrários.
2.6. A sincronização em monitores é feita por variáveis de condição e as operações `wait` e `signal`. Uma forma mais geral de sincronização seria termos uma primitiva única `waituntil`, que teria como parâmetro uma expressão booleana arbitrária, como, por exemplo:

```
waituntil x<0 or y+z < n
```

Neste caso, a primitiva `signal` não seria mais necessária. Este método, apesar de mais geral que o discutido, não é nunca utilizado. Por que? (Pense sobre a forma de implementação!).
2.7. Suponha que, num sistema de passagem de mensagens com *mailbox*, quando um processo tenta enviar para um *mailbox* cheio ou ler em um *mailbox* vazio, ao invés de ser bloqueado ele recebe um código de retorno indicando a condição do *mailbox*. O processo responde a esse código de erro tentando novamente a operação, continuamente, até que ele consiga. Este esquema pode levar a condições de disputa?
2.8. A implementação de monitores através de semáforos não utiliza uma lista ligada explícita, enquanto que a de semáforos através de monitores usa. Explique por que. (Pense sobre as diferenças entre semáforos e variáveis de condição).
2.9. Na solução do problema do jantar dos filósofos, por que razão a variável de estado é colocada em `faminto` na rotina `pegue_garfos`?
2.10. Quando falávamos das desvantagens do método da espera ocupada, citamos o caso de dois processos H e L, onde H ficava num *loop* eterno. Este problema ocorreria se fosse utilizado um escalonador *round-robin* ao invés de um de prioridade?
2.11. No escalonamento por *round-robin*, normalmente temos uma lista de todos os processos prontos para execução, com cada processo aparecendo uma única vez na lista. O que aconteceria se um processo aparecesse duas vezes? Você é capaz de indicar alguma razão para se permitir isto?
2.12. Explique por que o escalonamento em dois níveis é comumente utilizado.
2.13. Em um certo sistema, processos rodam em média por um tempo T antes de bloquearem aguardando E/S. Um chaveamento de processos requer um tempo S (que é considerado

desperdiçado). Para escalonamento *round-robin* com *quantum* Q, dê uma fórmula para a eficiência da CPU para cada um dos seguintes casos:
(a) $Q = \infty$
(b) $Q > T$
(c) $S < Q < T$
(d) $Q = S$
(e) $Q \approx 0$

2.14. Cinco *jobs* (designados pelas letras A até E) chegam num centro de computação aproximadamente no mesmo instante, mas na ordem A, B, C, D e E. Eles têm tempo de execução estimados em 10, 6, 2, 4 e 8, respectivamente. Suas prioridades são 3, 5, 2, 1 e 4, respectivamente, com 5 sendo a prioridade mais elevada. Para cada um dos algoritmos abaixo, calcule o tempo de resposta médio conseguido na execução desses *jobs*.
(a) Round-robin.
(b) Prioridade.
(c) Ordem de chegada.
(d) Menor *job* primeiro.

3. Entradas e Saídas

Uma das funções principais de um S.O. é controlar o acesso aos dispositivos de entrada e saída (abreviadamente: E/S ou I/O) e fornecer aos usuários uma interface simples com os mesmos. Na medida do possível, a interface oferecida para todos os dispositivos de E/S deve ser a mesma (o que garante a Independência de Dispositivo - ***device independency***).

3.1. Princípios de Hardware de E/S

Enfocaremos os dispositivos de E/S sob o ponto de vista de sua programação, que no entanto está muitas vezes ligada à sua operação interna.

3.1.1. Dispositivos de E/S

Dividimos os dispositivos de E/S em duas categorias:

dispositivos de bloco: armazenam a informação em blocos de tamanho fixo, cada um dos quais com o seu próprio endereço, podendo ser lido independentemente de todos os outros. Ex: discos.

dispositivos de caracteres: entrega ou aceita uma cadeia de caracteres, sem considerar qualquer estrutura de bloco. Ex: terminal e impressora.

Alguns dispositivos não se adaptam bem a nenhuma das duas categorias acima, por exemplo o relógio, que nem apresenta estrutura de bloco nem realiza transmissão de caracteres, realizando apenas uma interrupção a intervalos pré-definidos. Com isto vemos que a classificação acima não é exaustiva, servindo apenas como orientação para uma melhor compreensão de diversos dispositivos.

3.1.2. Controladores de dispositivos

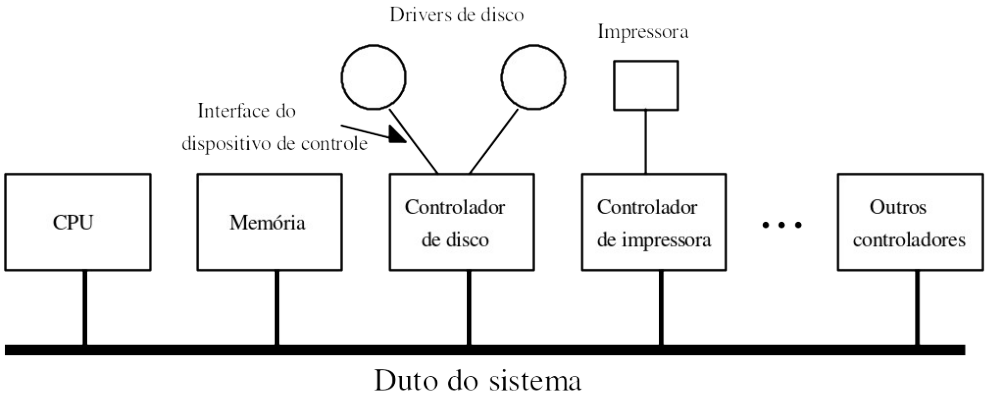
Em geral, os dispositivos estão constituídos em duas partes:

parte "mecânica", que é a que efetivamente executa as ações requeridas;

parte eletrônica, que controla os dispositivos, sendo por isso denominada controlador ou adaptador.

O S.O. trata com o controlador, e não com o dispositivo em si.

Para a comunicação da UCP com os dispositivos de E/S (isto é, com os seus controladores) são em geral usados **barramentos**. Um esquema de **barramento único** é indicado na figura abaixo:



Sistemas atuais em geral se utilizam de barramentos múltiplos e computadores especializados de E/S, chamados canais de E/S, que aliviam grande parte da "carga" da UCP com relação ao processamento de E/S.

A interface entre o controlador e o dispositivo é geralmente de baixo nível, cuidando de todos os detalhes do funcionamento do dispositivo, bem como da organização da informação de forma a ser reconhecida pela UCP.

Cada controlador possui alguns registradores, que são utilizados para comunicação com a UCP. Esta comunicação pode ser realizada de duas formas:

- **registradores no espaço de memória**: neste caso, os registradores dos controladores ocupam posições no espaço de endereçamento da memória;
- **registradores no espaço de E/S**: neste caso, os registradores dos controladores são incluídos num espaço separado de endereçamento, conhecido como **espaço de E/S**.

As operações de E/S são então realizadas através da escrita, pela UCP, de comandos nos registradores dos controladores e da leitura de resultados nesses mesmos registradores.

Acesso Direto à Memória

Vários controladores (especialmente de dispositivos de bloco) suportam uma técnica de transferência conhecida como **acesso direto à memória** (DMA, *direct memory access*). Para exemplificar a operação do DMA, vejamos como opera o acesso a disco com e sem a utilização dessa técnica.

Quando não existe DMA, o acesso a disco é feito da seguinte forma:

1. o controlador de disco lê, um a um, os bits de dados vindos do disco;
2. quando um bloco completo foi transferido, é feito um teste para verificar se houve erros (utilizando uma técnica de redundância conhecida como CRC);
3. se não houve erros, o controlador causa uma interrupção na UCP;
4. a UCP entra num *loop* de leitura, que lê um byte por vez do controlador

Como vemos, esse ciclo gasta um tempo considerável da UCP devido à necessidade desta permanecer num *loop* para a transferência de todos os dados provindos do disco. Para eliminar o tempo gasto pela UCP nesse *loop*, utilizamos a técnica de DMA, que faz com que o ciclo de transferência do disco fique como abaixo:

1. a UCP dá ao controlador dados sobre o setor a ler, endereço de memória onde ela quer que a informação seja colocada, e número de bytes que devem ser transferidos
2. após o controlador ler o bloco e verificar os erros, como indicado anteriormente, ele automaticamente **copia todo o bloco na posição de memória especificada** e só depois de terminar a cópia gera uma interrupção para avisar a UCP do final da transferência;

3. quando o S.O. quiser utilizar os dados, estes já se encontram na posição correta da memória, não sendo necessário utilizar tempo de UCP para a transferência.

3.2. Princípios de Software de E/S

Devemos procurar organizar o *software* de E/S em camadas, de forma que as camadas mais baixas ocultem as particularidades do *hardware* às camadas mais altas, e estas últimas se ocupem de apresentar ao usuário uma interface clara e agradável.

3.2.1. Objetivos do software de E/S

Os conceitos importantes que devem ser considerados num projeto de *software* para E/S são os seguintes:

1. independência de dispositivo: isto é, deve ser possível escrever programas que possam ser utilizados, sem alteração, para dispositivos diversos, como discos rígidos, cartões de memória, *pendrives*, e mesmo terminais de vídeo e linhas seriais. O S.O. é que deve cuidar dos problemas causados pelas diferenças entre os dispositivos, e não o usuário;
2. denominação uniforme: os nomes dados aos dispositivos devem ser independentes do tipo do mesmo. Por exemplo, no UNIX todos os dispositivos e arquivos são acessados através de um nome em um diretório;
3. tratamento de erros: deve-se tratar os erros em níveis tão mais próximos do *hardware* quanto possível, de forma que os níveis superiores somente se ocupam com erros que não podem ser tratados pelos níveis inferiores;
4. transmissão síncrona (bloqueante) × assíncrona (interrupções): na transmissão **assíncrona**, a UCP pede uma transferência e em seguida vai realizar outras operações. Quando a transferência está pronta, uma interrupção é gerada para informar a UCP de que a operação foi completada. Ao se escrever programas, é mais simples que as transmissões sejam consideradas **bloqueantes**, de forma que, ao pedir uma transferência, o programa pára até que ela esteja pronta (por exemplo, num `read`). O S.O. deve cuidar de fazer com que uma transferência assíncrona pareça ao usuário como síncrona;
5. dispositivos compartilhados × dedicados: alguns dispositivos (p.ex.: discos) podem ser compartilhados entre diversos usuários sem problema, podendo vários usuários possuírem arquivos abertos no mesmo. O mesmo já não acontece com outros (p.ex.: impressora), que devem ser utilizados por um único usuário por vez. O S.O. deve ser capaz de cuidar tanto de dispositivos compartilhados como de dispositivos dedicados, de uma forma que evite problemas em qualquer dos casos.

Os objetivos indicados acima podem ser atingidos de um modo abrangente e eficiente estruturando-se o *software* de E/S em quatro camadas:

1. tratadores de interrupções;
2. condutores de dispositivos (**device drivers**);
3. *software* de E/S independente de dispositivo;
4. *software* em nível de usuário.

Vejamos as atribuições de cada uma dessas camadas.

3.2.2. Tratadores de interrupção

Devido à extrema dependência do *hardware*, as interrupções devem ser tratadas nos níveis mais baixos do S.O., de forma que a menor parte possível do sistema tome conhecimento da existência de interrupções.

O método tradicional de trabalho é fazer com que os condutores de dispositivos (*device drivers*) sejam bloqueados quando executam um comando de E/S. Quando a interrupção correspondente ocorre, a rotina de interrupção desbloqueia esse processo (através de um `V` em semáforo, ou um `signal` em variável de condição ou da transmissão de uma mensagem, dependendo do tipo de sincronização utilizado pelo S.O.).

3.2.3. Condutores de dispositivos (device drivers)

O condutor de dispositivo contém todo o código dependente do dispositivo. Cada condutor de dispositivo maneja um tipo de dispositivo ou, no máximo, uma classe de dispositivos correlacionados.

O condutor de dispositivo envia os comandos aos controladores (através dos registradores corretos) e verifica que esses comandos sejam corretamente executados, sendo a única parte do S.O. que tem conhecimento do controlador utilizado. Em termos gerais, a tarefa do condutor de dispositivo é aceitar comandos do *software* independente de dispositivo e fazer com que sejam executados, devendo transformar o pedido dos termos abstratos em que é realizado para termos concretos. Por exemplo, no condutor de disco, verificar onde fisicamente o bloco requisitado se encontra, verificar o estado do motor (ligado ou desligado), posição da cabeça de leitura, etc.

Ele então determina os comandos que devem ser enviados ao controlador do dispositivo e os envia. A partir desse ponto temos duas situações possíveis:

1. o controlador leva algum tempo até terminar a execução do comando, caso em que o condutor de dispositivo se bloqueia; ou
2. o controlador executa o comando automaticamente, não sendo necessário o bloqueio.

No primeiro caso, o condutor bloqueado será acordado por uma interrupção proveniente do controlador.

Os dados obtidos com a operação são então passados ao *software* independente de dispositivo.

3.2.4. Software independente de dispositivo

A separação exata entre os condutores e o *software* independente de dispositivo é variável com o S.O. As funções geralmente realizadas pela parte do *software* independente de dispositivo são as seguintes:

- interfaceamento uniforme para os condutores de dispositivo: é a função básica, compreendendo a execução de funções comuns a todos os dispositivos de E/S;
- denominação dos dispositivos: consiste no mapeamento dos nomes simbólicos nos condutores corretos (p.ex.: em UNIX `/dev/tty0` especifica um arquivo especial que contém o número de dispositivo que é utilizado para localizar o condutor apropriado para terminais, e mais outro número utilizado para especificar a unidade a ser utilizada, no caso o console);
- proteção dos dispositivos: em microprocessadores, em geral o usuário pode fazer o que desejar com os dispositivos de E/S, não existindo nenhuma proteção implícita. Já em computadores grandes, o acesso a dispositivos de E/S é proibido aos usuários. Em UNIX, o acesso a dispositivos de E/S é protegido pelos bits `rwx`, como qualquer arquivo;

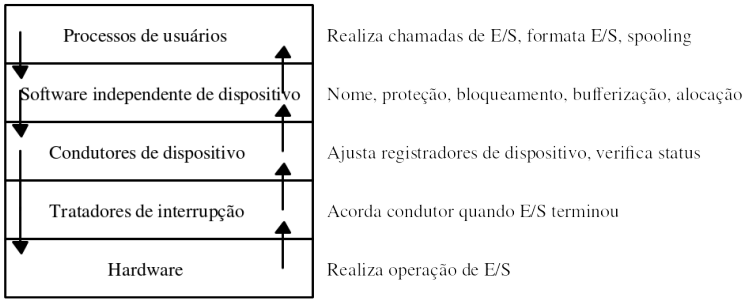
- fornecimento de um tamanho de bloco independente do dispositivo: para dispositivos de bloco, devemos fazer com que todos os presentes no sistema apareçam ao usuário com o mesmo número de blocos. Para dispositivos de caracter, alguns apresentam um caracter por vez (p.ex.: terminais), enquanto que outros apresentam-nos em conjuntos de vários caracteres (p.ex.: leitora de cartões). Essas diferenças devem ser ocultadas;
- bufferização: deve existir bufferização para se poder lidar com situações como as seguintes:
  a) dispositivos de bloco requerem o acesso a blocos completos, enquanto que o usuário pode requerer acesso a unidades menores;
  b) em dispositivos de caracteres, o usuário pode tentar escrever mais rápido do que o dispositivo pode ler, ou o dispositivo pode enviar antes que o usuário esteja pronto para ler;
- alocação de memória em dispositivos de bloco: a descoberta de quais os blocos livres e quais os ocupados em um dispositivo de bloco não é dependente do dispositivo, e pode, portanto, ser realizada neste nível do S.O.;
- alocação e liberação de dispositivos dedicados: pode ser tratada também nesta parte do S.O. Normalmente isto é feito com chamadas de sistema do tipo `open` (para pedir acesso) e `close` (para liberar o acesso);
- informe de erros: a maioria dos erros é altamente dependente do *hardware* e deve, portanto, ser tratada nos condutores de dispositivos. No entanto, após um condutor informar um erro, o tratamento desse erro é independente do dispositivo, sendo em geral o erro informado ao processo que pediu o acesso. No caso de um erro que possa comprometer completamente o funcionamento do sistema, entretanto, uma mensagem deve ser enviada e o sistema terminado.

3.2.5. Software de E/S no espaço de usuário

Uma parte do *software* de E/S consiste em rotinas de biblioteca ligadas com programas de usuário, e mesmo programas inteiros rodando fora do modo *kernel*. Chamadas de sistema, incluindo-se as de E/S, são feitas geralmente por rotinas de biblioteca. Muitas dessas rotinas não fazem nada mais do que colocar os parâmetros no local apropriado para a chamada de sistema e realizar a chamada. Outras entretanto fazem um trabalho real, em geral de formatação de dados (p.ex.: `printf`).

Temos também os sistemas de *spooling*, que consistem em uma forma de tratar dispositivos de E/S dedicados em sistemas de multiprogramação. O exemplo típico de dispositivo que utiliza *spool* é a impressora. A impressora poderia ser tratada através da utilização de `open` e `close`. No entanto, considere o que ocorreria caso um usuário realizasse `open` e não utilizasse a impressora por horas! Dessa forma, é criado um processo especial, chamado `daemon`, que é o único a ter permissão de acesso à impressora, e um diretório de *spool*. Quando um processo quer imprimir um arquivo ele o coloca no diretório de *spool*, e então o *daemon* o imprime, quando possível.

O sistema de E/S pode ser resumido como na figura seguinte:



3.3. Deadlock

Um problema com a técnica de *spool* é que o arquivo inteiro precisa ser escrito no disco, antes que a saída possa se iniciar (pense no que ocorreria caso um *daemon* iniciasse a impressão antes que o arquivo seja completamente gerado e o processo que gera o arquivo para por algumas horas antes de terminar a geração!).

Em alguns casos, a saída gerada pode ser muito grande, o que inviabiliza que todos os dados sejam escritos no disco antes de serem enviados à unidade. A única solução nesses casos é não utilizar *spool* nesse dispositivo, mas fazer com que os usuários tenham acesso exclusivo ao mesmo quando precisarem.

No entanto, esta solução pode gerar *deadlock*. Suponha a seguinte situação:

- processo A pede unidade de fita magnética e recebe
- processo B pede *plotter* e recebe
- processo A pede *plotter* sem liberar a fita magnética
- processo B pede fita magnética sem liberar *plotter*

Neste caso percebemos que tanto o processo A como o processo B são postos para dormir, situação na qual permanecerão indefinidamente.

3.3.1. Recursos

Chamaremos de recurso qualquer objeto ao qual deva ser dado acesso exclusivo para cada processo. Recursos podem ser dispositivos de *hardware* ou trechos de informação. Para alguns recursos, diversas instâncias idênticas podem estar disponíveis (p.ex.: diversas unidades de fita), caso em que cada uma pode ser utilizada para satisfazer um pedido distinto. Para a utilização de um recurso temos, então a seguinte seqüência, a ser realizada pelo processo:

1. Requisitar o recurso;
2. Utilizar o recurso;
3. Liberar o recurso.

Se um recurso não estiver disponível quando for requisitado, o processo que o requisitou é forçado a aguardar, sendo dois métodos utilizados:

1. o processo é bloqueado, e acordado quando o recurso estiver disponível; ou
2. é enviado um código de erro ao processo, indicando que a requisição falhou, sendo então que o próprio processo deve decidir a ação a tomar (p.ex.: aguardar algum tempo e pedir novamente).

Em UNIX, os recursos são requisitados através de open, utilizando-se o segundo dos métodos indicados acima.

3.3.2. Deadlock

Definição formal:

Deadlock: ocorre quando cada processo de um conjunto de processos está esperando por um evento que apenas outro processo do mesmo conjunto pode causar.

Em muitos casos, o evento esperado é a liberação de um recurso qualquer, isto é, cada membro do conjunto está esperando pela liberação de um recurso que apenas outro membro do conjunto pode liberar. O número de processos ou recursos envolvidos nessa situação não é importante.

Quatro condições são necessárias para que ocorra *deadlock*:

1. **Exclusão mútua**: cada recurso ou está associado a exatamente um processo ou está disponível;
2. **Posse e espera** (*hold and wait*): um processo que já possui algum recurso pode requisitar outros e aguardar por sua liberação;
3. **Não existe preempção**: recursos dados a um processo não podem ser tomados de volta. Eles precisam ser explicitamente liberados pelo processo;
4. **Espera circular**: Deve haver uma cadeia circular de dois ou mais processos, cada um dos quais aguardando um recurso em posse do próximo membro da cadeia.

Podemos modelar a distribuição dos recursos e das requisições dos mesmos através de um grafo com dois tipos de nós:

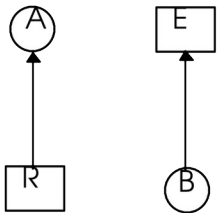
circular: representa um processo;

retangular: representa um recurso.

O significado dos arcos entre os nós é o seguinte:

- arco de um recurso para um processo: indica que esse recurso está entregue ao processo apontado;
- arco de um processo para um recurso: indica que o processo está bloqueado, aguardando o recurso.

Veja as figuras abaixo, onde A e B são processos e R e E são recursos:



Analisemos agora um caso mais complexo. Suponhamos a existência de três processos A, B e C, e de três recursos R, S, e T. Suponhamos ainda para cada processo a seguinte ordem de requisição e liberação de recursos:

A:
pede R
pede S
libera R
libera S

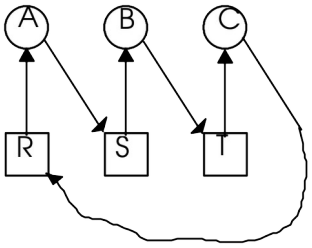
B:
pede S
pede T
libera S
libera T

C:
pede T
pede R
libera T
libera R

Se o S.O. executa cada um dos processos por vez, esperando que um processo termine antes de executar o outro, então claramente não teremos *deadlock*, mas também não teremos nenhum paralelismo. Suponhamos então que queremos realizar o escalonamento dos processos em um método de *round-robin*. As requisições de recursos podem então ocorrer da seguinte forma:

A pede R
B pede S
C pede T
A pede S
B pede T
C pede R

Neste ponto, o grafo correspondente será como indicado na figura abaixo:

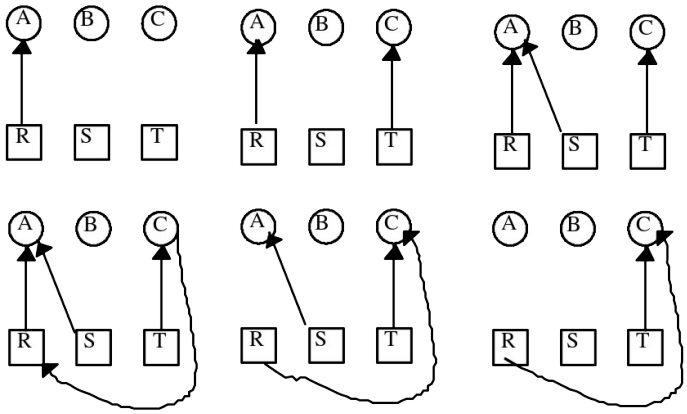


Vemos então que ocorreu uma cadeia circular de processos e recursos, indicando um *deadlock*.

Entretanto, como o S.O. tem liberdade de executar os processo, e portanto liberar os recursos na ordem que julgar mais adequada, se ele soubesse da possibilidade de ocorrência de *deadlock* na situação acima, poderia deferir, por exemplo, a execução do processo B, fazendo com que o mesmo aguardasse para iniciar quando já não houvesse perigo de *deadlock*. Assim a ordem de execução poderia ser a seguinte:

A pede R
C pede T
A pede S
C pede R
A libera R
A libera S

E vemos então que não haverá *deadlock*. Note que a partir deste ponto, B pode iniciar a receber os recursos. As figura abaixo indica a evolução dos pedidos indicados acima.



Em geral são utilizadas quatro estratégias para se tratar com o *deadlock*:

1. ignorar o problema;
2. detecção e recuperação;
3. prevenção, negando uma das quatro condições necessárias;
4. evitamento dinâmico, por uma alocação cuidadosa dos recursos.

Vejamos as técnicas e implicações envolvidas.

3.3.3. O "algoritmo da avestruz"

A estratégia mais simples para lidar com o *deadlock* é a mesma da avestruz frente a um perigo: colocar a cabeça em um buraco na areia e fingir que o problema não existe.

Apesar de teoricamente inaceitável, esta solução é a mais utilizada devido aos seguintes fatores:

- baixa probabilidade de ocorrência de *deadlock*, muito menor do que da ocorrência de quebra do sistema por erros no próprio S.O., em compiladores ou no *hardware*.
- o alto preço pago pelos usuários para se prevenir dos *deadlock*s, como veremos mais detalhadamente a seguir.

O UNIX utiliza este método.

3.3.4. Detecção e recuperação

Nesta técnica, o S.O. apenas monitora as requisições e liberações de recursos, através da manutenção de um grafo de recursos, que é constantemente atualizado, e onde se verifica a ocorrência de ciclos. Se houver algum ciclo, um dos processos do ciclo deve ser morto (terminado à força). Se ainda permanecer o ciclo, outro processo deve ser morto, e assim sucessivamente, até que o ciclo seja quebrado.

É uma técnica utilizada em computadores grandes, geralmente em *batch*, onde um processo pode ser morto e mais tarde reinicializado. Deve-se, entretanto, ter cuidado de que qualquer arquivo modificado pelo processo morto deve ser restaurado ao seu estado original antes de iniciar o processo novamente.

3.3.5. Prevenção de deadlock

Consiste em impor restrições aos processos de forma que o *deadlock* seja impossível. Se garantirmos que uma das quatro condições necessárias (veja item 3.3.2.) nunca é satisfeita, então o *deadlock* é impossível. Analisemos quais as possibilidades de eliminar as condições:

1) Exclusão mútua: como vimos, para alguns recursos é necessária a exclusão mútua, pois não pode haver acesso por dois processos simultaneamente e nem sempre a técnica de *spool* pode ser empregada. Portanto esta condição não pode ser eliminada.
2) Posse e espera: Uma forma de negar esta condição é fazer com que cada processo requisite todos os recursos de que irá necessitar antes de iniciar a execução. Se todos os recursos estiverem disponíveis então ele é executado. Caso contrário, o processo deve aguardar. Um problema com este método é que muitos processos não sabem com antecedência quais os recursos de que irão necessitar até começar a execução. Outro problema é que os recursos não serão utilizados otimamente (Você sabe indicar por que?). Um modo diferente de quebrar esta condição é fazer com que, para pedir um novo recurso, o processo deva liberar temporariamente todos aqueles que possui, e somente se conseguir o novo recurso é que pode pegar os anteriores de volta.
3) Não preempção: é fácil notar que tomar um recurso de um processo para entregá-lo a outro é gerador de confusão. Portanto, esta condição não pode ser quebrada.
4) Espera circular: Vejamos alguns métodos de evitar a espera circular (isto é, evitar que se formem ciclos fechados no grafo de recursos):
   - cada processo só pode ter um recurso por vez e, se desejar outro, deve liberar o que possui. Isto impossibilitaria coisas simples como cópia de um dispositivo para outro;

- é dada uma numeração global a todos os recursos, e os processos só podem requisitar recursos em ordem numérica estritamente crescente. O problema aqui é que é difícil encontrar uma numeração que satisfaça a maioria das possíveis condições de acesso.

3.3.6. Evitamento dinâmico de deadlock

Neste método procura-se evitar o *deadlock* sem impor restrições aos processos. Isto é possível desde que alguma informação sobre a utilização de recursos pelo processo esteja disponível antecipadamente ao S.O. É o que analisaremos agora.

Algoritmo do banqueiro para um único recurso

O algoritmo do banqueiro, introduzido por Dijkstra em 1965 evita *deadlock*s. Consiste em simular as decisões de um banqueiro no empréstimo de certa quantia de dinheiro, sujeito a certas condições especiais, como veremos.

Na figura abaixo vemos a representação de quatro clientes (A, B, C e D) que têm linhas de crédito com um banqueiro. Cada um especificou o número máximo de unidades de crédito que irá precisar, mas eles não precisam de todas elas imediatamente, de forma que o banqueiro reservou 10 unidades para atender todos os pedidos (que totalizam 22 unidades).

Nome	Usado	Máximo
A	0	6
B	0	5
C	0	4
D	0	7

disponível: 10

Considera-se que:

- cada cliente pode, em cada momento, realizar um pedido desde que este não faça com que ultrapasse seu limite de crédito
- o banqueiro pode escolher livremente o momento de atendimento dos pedidos realizados, apenas se comprometendo em atendê-lo em um tempo finito
- após o recebimento de todo o crédito que especificou inicialmente (isto é, o seu valor de Máximo), o cliente se compromete a devolver, dentro de um tempo finito, tudo o que lhe foi emprestado

O banqueiro deve analisar cuidadosamente, portanto, cada pedido de recursos, para garantir que não entre numa situação da qual não possa sair. Vejamos um exemplo. Chamamos de **estado** a situação dos empréstimos, bem como dos recursos ainda disponíveis em um dado momento. Na figura abaixo temos um exemplo:

Nome	Usado	Máximo
A	1	6
B	1	5
C	2	4
D	4	7

disponível: 2

Um estado é dito **seguro** se existe uma seqüência de outros estados que leva a que todos os clientes recebam seus empréstimos até seus limites de crédito. O estado da figura acima é seguro, pois o banqueiro pode emprestar as duas unidades a C, pegar a devolução do mesmo (4 unidades) e emprestar a B, pegar as 5 unidades de volta e emprestar a A, pegar as 6 unidades de volta e emprestar 3 delas a D, pegando então as 7 unidades de volta, garantindo que todos os clientes foram atendidos.

Suponhamos entretanto que, na mesma situação da figura anterior, o cliente B fizesse um pedido de mais uma unidade e o banqueiro atendesse. Teríamos então a situação da figura seguinte:

Nome	Usado	Máximo
A	1	6
B	2	5
C	2	4
D	4	7

disponível: 1

Como é fácil verificar, este não é um estado seguro, pois não há forma de o banqueiro garantir o atendimento de todos os pedidos, o que pode vir a gerar um *deadlock*. Importante notar que ainda não estamos em *deadlock*, pois os valores de máximo não precisam necessariamente ser atingidos. Por exemplo, o cliente D pode decidir que não precisa mais de novos recursos, e devolver os 4 que já pediu ao banqueiro. Desta forma estaríamos novamente numa situação segura. No entanto, o banqueiro não pode contar com isso, pois não tem formas de saber quando um cliente irá necessitar ou não do máximo que especificou.

O algoritmo do banqueiro portanto consiste em, para cada pedido que chega, verificar se conceder o mesmo leva a um estado seguro ou não. Se levar a um estado seguro o cliente é satisfeito. Para verificar se o estado é seguro ou não ele deve verificar se o disponível é suficiente para atender o cliente mais próximo de seu máximo. Se for, finge que esse cliente já devolveu tudo que possuía, marca o mesmo como terminado, e verifica se pode atender o cliente mais próximo do máximo entre os que restam. Se ele puder levar a bom termo esse processo, atendendo (em sua análise) todos os clientes, então o estado é seguro.

O algoritmo do banqueiro para vários recursos

Neste caso, montamos duas matrizes, uma de recursos entregues e outra de recursos ainda necessários, sendo que cada linha das matrizes corresponde a um processo e cada coluna corresponde a um tipo de recurso, como na figura abaixo:

P	F	T	I	R
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Recursos alocados

P	F	T	I	R
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Recursos ainda necessários

E=(6342)

P=(5322)

D=(1020)

Legenda:

P: Processos

F: unidades de fita

T: Plotters

I: Impressoras

R: CD-ROM

Temos também três vetores E, P e D que indicam:
E → recursos existentes;
P → recursos possuídos por algum processo;
D → recursos disponíveis.

Como no caso de um único recurso, os processos precisam avisar com antecedência das suas necessidades de recursos. Para descobrir se um estado é seguro ou não procedemos, então, da seguinte forma:

1. Procuramos uma linha L cujos recursos ainda necessários sejam todos menores que os correspondentes valores no vetor D. Se não existir uma linha assim, o sistema provavelmente entrará em *deadlock*, e o estado não é seguro.
2. Assumir que o processo da linha L requereu todos os recursos e terminou, liberando todos os recursos que estava ocupando.
3. Pepetir os passos 1 e 2 até que todos os processos sejam marcados como terminados. Se isto for conseguido o estado é seguro.

O problema com este algoritmo é que os processos raramente sabem com antecedência a quantidade máxima de recursos que necessitarão. Além disso, o número de processos no sistema varia dinamicamente, o que invalida as análises realizadas.

Não é conhecido nenhum algoritmo para cuidar do caso geral, sendo que apenas se utilizam alguns em aplicações de propósito específico.

Exercícios

3.1. O que é independência de dispositivo?

3.2. Por que os arquivos de saída para a impressora são normalmente gravados em um diretório de *spool* antes de serem enviados para a impressora?

3.3. Em qual das 4 camadas do *software* de E/S é feita cada uma das seguintes operações:
a) cálculo da trilha, setor e cabeça, numa leitura do disco;
b) escrita de comandos nos registradores dos controladores de dispositivos;
c) verificação se o usuário tem permissão para utilizar um dispositivo;
d) converter inteiros em binário para ASCII numa impressão.

3.4. Suponha que, na análise que realizamos de três processos utilizando três recursos, o processo C requisitasse o recurso S ao invés do R. Haveria um *deadlock*? E se ele requisitasse S ao invés de T?

3.5. Suponha um computador com 6 unidades de fita e *n* processos competindo por elas. Para quais valores de *n* o sistema é livre de *deadlock*?

3.6. Pode um sistema estar em um estado que não é nem *deadlock* nem um estado seguro? Se sim, dê um exemplo, se não, prove que uma terceira possibilidade não existe.

3.7. Um sistema distribuído utilizando *mailboxes* tem duas primitivas de comunicação entre processos, `send` e `receive`. A última primitiva especifica um processo do qual receber, e bloqueia se nenhuma mensagem daquele processo está disponível, mesmo que mensagens de outros processos estejam esperando. Não existem recursos compartilhados, mas os processos necessitam se comunicar freqüentemente sobre outros assuntos. O *deadlock* é possível? Discuta.

3.8. Em um sistema eletrônico de transferência de fundos, existem centenas de processos semelhantes que atuam da seguinte maneira: cada processo lê uma linha de entrada que especifica o total de dinheiro, a conta a ser creditada, e a conta a ser debitada. Então ele trava ambas as contas (isto é, impede que as mesmas sejam acessadas por outros processos) e transfere o dinheiro, liberando as contas quando termina. Com muitos processos rodando em

paralelo, existe um perigo real de *deadlock*. Imagine um esquema para evitar *deadlock*. Não libere uma conta até que tenha sido completada a transação (i.e.: não são permitidas soluções em que uma conta é travada e liberada em seguida se a outra conta já estiver travada).

3.9. Qual a complexidade do algoritmo do banqueiro para múltiplos recursos? Isto é, sendo m o número de classes de recursos e n o número de processos, e considerando que m e n são suficientemente grandes, o número de operações que precisam ser executadas para decidir se uma requisição deve ser atendida é aproximadamente $m^a n^b$. Encontre os valores de a e b.

4. Gerenciamento de memória

Trataremos agora da parte do S.O. conhecida como **gerenciador de memória**, que é o responsável por cuidar de quais partes da memória estão em uso, quais estão livres, alocar memória a processos quando eles precisam, liberar quando eles não mais necessitam e gerenciar a troca (*swapping*) dos processos entre memória principal e disco, quando a memória principal não é suficientemente grande para manter todos os processos.

4.1. Gerenciamento de memória sem troca ou paginação

Troca e paginação são métodos utilizados de movimentação da memória para o disco e vice-versa durante a execução dos processos. Entretanto, antes de estudar esses métodos, estudaremos o caso mais simples em que não é realizada nem troca nem paginação.

4.1.1. Monoprogramação sem troca ou paginação

Neste caso, temos um único processo executando por vez, de forma que o mesmo pode utilizar toda a memória disponível, com exceção da parte reservada ao sistema operacional, que permanece constantemente em local pré-determinado da memória. O S.O. carrega um programa do disco para a memória, executa-o, e em seguida aguarda comandos do usuário para carregar um novo programa, que se sobreporá ao anterior.

4.1.2. Multiprogramação e utilização de memória

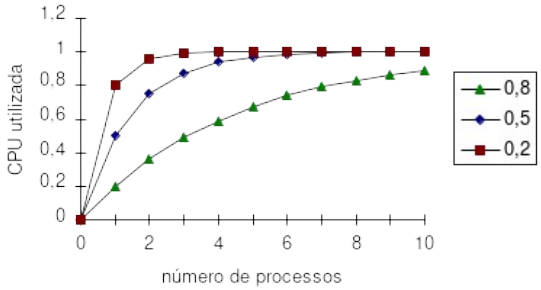
Apesar de largamente utilizada em microcomputadores antigos, a monoprogramação praticamente não é mais utilizada. Algumas razões para isso são as seguintes:

- Muitas aplicações são mais facilmente programáveis quando as dividimos em dois ou mais processos.
- Os grandes computadores em geral oferecem serviços interativos simultaneamente para diversos usuários. Neste caso, é impossível de se trabalhar com um único processo na memória por vez, pois isso representaria grande sobrecarga devido à constante necessidade de chavear de um processo para outro (o que, com apenas um processo por vez na memória representaria constantemente estar-se lendo e escrevendo no disco).
- É necessário que diversos processos estejam “simultaneamente” em execução, devido ao fato de que muitos deles estão constantemente realizando operações de E/S, o que implica em grandes esperas, nas quais, por questão de eficiência, a UCP deve ser entregue a outro processo

Modelamento de multiprogramação

Podemos apresentar de forma breve, uma idéia da importância da existência de diversos processos simultaneamente na memória em face das operações de E/S da seguinte forma:

- Suponhamos que cada processo toma uma fração *p* de seu tempo aguardando por E/S
- A probabilidade de que *n* processos estejam esperando por E/S simultaneamente é, portanto, de p^n
- A utilização da UCP é, portanto, $1-p^n$
- A figura abaixo mostra a utilização da UCP como uma função do grau de multiprogramação (*n*), para três valores de *p* (0.2, 0.5 e 0.8):



Vemos então que se os processos gastam 80% de seu tempo aguardando por E/S, então devemos ter mais de 10 processos na memória para garantir que a UCP perca menos de 10% de seu tempo.

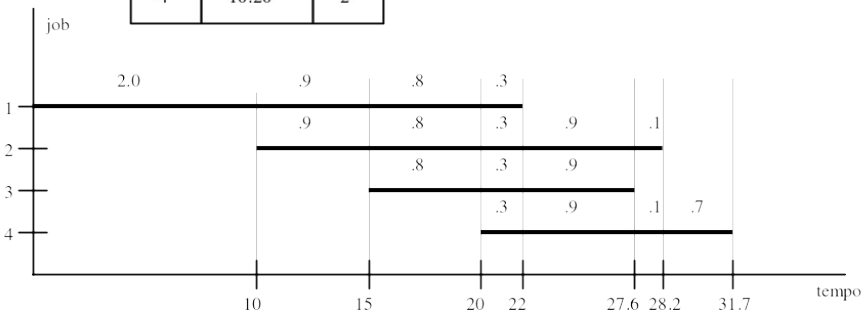
É obvio que o modelo acima é extremamente simplificado, mas dá uma boa idéia do formato geral da relação entre o grau de multiprogramação e o grau de utilização da UCP.

Análise da performance de sistemas multiprogramados

Como exemplo, vejamos o que ocorre com um sistema processando 4 *jobs*, com requisições de tempo de UCP e instantes de chegada conforme indicado na figura abaixo. Consideremos ainda que nenhum processo tem privilégio sobre o outro, de modo que será utilizado um esquema de *round-robin* para o escalonamento. A figura mostra quais os processos que estão executando em cada momento e a utilização de UCP. Considera-se que os *jobs* permanecem 80% de tempo aguardando operações de E/S, o que significa que, para cada minuto passado, 12 segundos o processo utiliza a UCP e durante os outros 48 segundos permanece bloqueado aguardando operações de E/S (por exemplo, o *job* 1, para executar os 4 minutos de UCP de que necessita permanecerá na memória por 20 minutos).

Job	Tempo de chegada	Tempo cpu necessário
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

N. de Processos	1	2	3	4
CPU inativa	.80	.64	.51	.41
CPU ocupada	.20	.36	.49	.59
CPU/processo	.20	.18	.16	.15



4.1.3. Multiprogramação com partições fixas

Já que percebemos a importância da existência de diversos processos na memória, devemos agora analisar de que forma este objetivo pode ser conseguido. A forma mais simples é dividir a memória existente em n partições fixas, possivelmente diferentes. Essas partições poderiam ser criadas, por exemplo, pelo operador, ao inicializar o sistema.

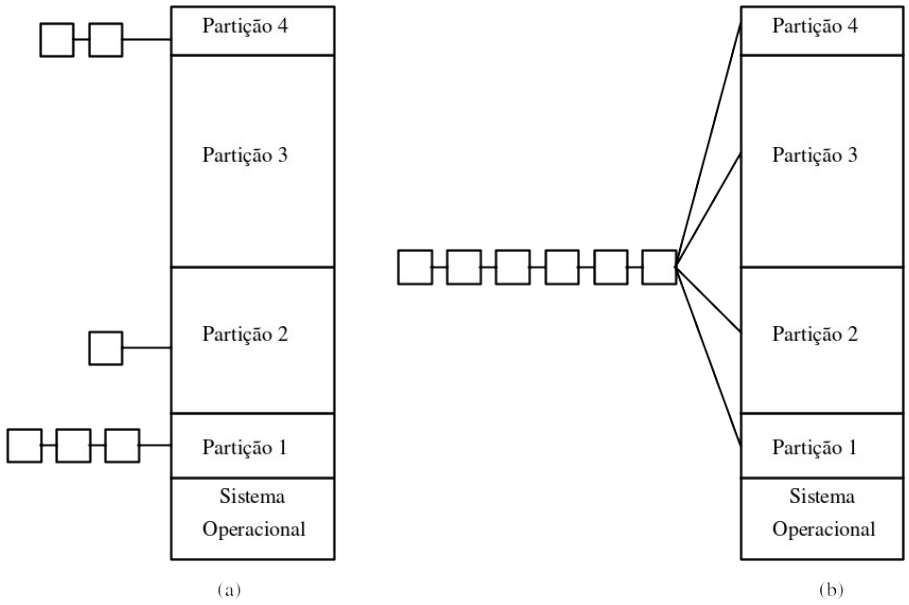
Uma forma de tratar com essas partições seria então a seguinte:

(i). cria-se uma fila para cada partição existente;

(ii). cada vez que um processo é iniciado, ele é colocado na fila da menor partição capaz de o executar;

(iii). os processos em cada partição são escolhidos de acordo com alguma forma de política (p.ex.: primeiro a chegar é atendido antes).

Este método pode ser simbolizado como na figura abaixo (a):



Uma desvantagem óbvia desse esquema é a de que pode ocorrer que uma partição grande esteja sem utilização, enquanto que diversos processos estão aguardando para utilizar uma partição menor (p.ex.: na figura acima temos três processos aguardando pela partição 1, enquanto que a partição 3 está desocupada).

Podemos resolver esse problema da seguinte forma:

(i). estabelecemos apenas uma fila para todas as partições;

(ii). quando uma partição fica livre, um novo processo que caiba na partição livre é escolhido e colocado na mesma, conforme indicado na figura anterior (b).

O problema que surge aqui é a forma de escolha implícita em (ii). Se a partição livre for entregue para o primeiro processo da fila, pode ocorrer de que uma partição grande seja entregue a um processo pequeno, o que não é desejável (pois implica desperdício de memória). Por outro lado, se percorremos a fila procurando o maior processo aguardando que caiba na partição, isto representará servir melhor os processos grandes em detrimento dos pequenos.

Relocação e Proteção

Relocação e proteção são dois problemas introduzidos pela multiprogramação, e que precisam ser resolvidos. A necessidade da relocação pode ser vista nas figuras anteriores, onde fica claro que processos diferentes executam em posições de memória diferentes e, portanto, com endereços diferentes.

Suponha um sistema com partições de 100k e uma rotina em um programa que, ao término da ligação dos módulos deste, é colocada na posição 100 em relação ao início do programa. É claro que se esse programa for ser executado na partição 1, todas as chamadas dessa rotina devem ser enviadas para a posição de memória 100k+100. Se o programa for executado na partição 2, as chamadas para essa mesma rotina devem ser enviadas para a posição 200k+100.

Uma possível solução é modificar as instruções conforme o programa é carregado na memória. Desta forma, quando o S.O. carrega o programa, adiciona a todas as instruções que se referenciem a endereços, o valor do ponto inicial de carga do programa (i.e., o início da partição escolhida para o mesmo). Esta solução exige que o ligador (*linker*) coloque no início do código binário do programa uma tabela (ou uma lista) que apresente as indicações das posições no programa que devem ser modificados no carregamento.

Esta solução entretanto não resolve o problema da proteção, pois nada impede que um programa errado ou malicioso leia ou altere posições de memória de outros usuários (dado que as referências são sempre a posições absolutas de memória).

Uma solução para isto, adotada no IBM 360, foi a de dividir a memória em unidades de 2k bytes e associar um código de proteção de 4 bits a cada uma dessas regiões. Durante a execução de um processo, o PSW (*program status word)* contém um código de 4 bits, que é testado com todos os acessos à memória realizados pelo processo, e gera uma interrupção se tentar acessar uma região de código diferente.

Uma solução alternativa tanto para o problema da relocação como para o da proteção é a utilização de registradores de base e limite. Sempre que um processo é carregado na memória, o S.O. ajusta o valor do registrador de base de acordo com a disponibilidade de memória. Este registrador é utilizado de forma que, sempre que um acesso é realizado na memória pelo processo, o valor do registrador de base é automaticamente somado, o que faz com que o código do programa em si não precise ser modificado durante o carregamento. O registrador de limite é utilizado para determinar o espaço de memória que o processo pode executar. Todo acesso realizado pelo processo à memória é testado com o valor do registrador limite, para verificar se esse acesso está se realizando dentro do espaço reservado ao processo (caso em que ele é válido) ou fora do seu espaço (acesso inválido).

Uma vantagem adicional do método dos registradores base e limite é o de permitir que um programa seja movido na memória, mesmo após já estar em execução. No método anterior isto não era possível sem realizar todo o processo de alteração dos endereços novamente.

4.2. Troca (swapping)

Num sistema em *batch*, desde que se mantenha a UCP ocupada o máximo de tempo possível, não existe razão para se complicar o método de gerenciamento da memória. Entretanto, num sistema de *time-sharing*, onde muitas vezes existe menos memória do que o necessário para manter todos os processos de usuário, então é preciso que uma parte dos processos sejam temporariamente mantidos em disco. Para executar processos que estão no disco, eles devem ser enviados para a memória, o que significa retirar algum que lá estava. Este processo é denominado troca (*swapping*), e é o que passaremos a estudar.

4.2.1. Multiprogramação com partições variáveis

A utilização de partições fixas para um sistema com *swapping* é ineficiente, pois implicaria em que grande parte do tempo as partições estariam sendo utilizadas por processos muito pequenos para as mesmas.

A solução para isso é tratarmos com partições variáveis, isto é, partições que variam conforme as necessidades dos processos, tanto em tamanho, como em localização, como em número de partições. Isto melhora a utilização de memória mas também complica a alocação e liberação de memória.

É possível combinar todos os buracos formados na memória em um único, o que é conhecido como compactação de memória, mas isto é raramente utilizado, devido à grande utilização de UCP requerida.

Um ponto importante a considerar é quanta memória deve ser alocada a um processo, quando o mesmo é iniciado. Se os processos necessitam de uma certa quantidade pré-fixada e invariante de memória, então basta alocar a quantidade necessária para cada processo ativo e o problema está resolvido. Entretanto, muitas vezes os processos necessitam mais memória durante o processamento (alocação dinâmica de memória). Neste caso, quando um processo necessita de mais memória, podem ocorrer dois casos:

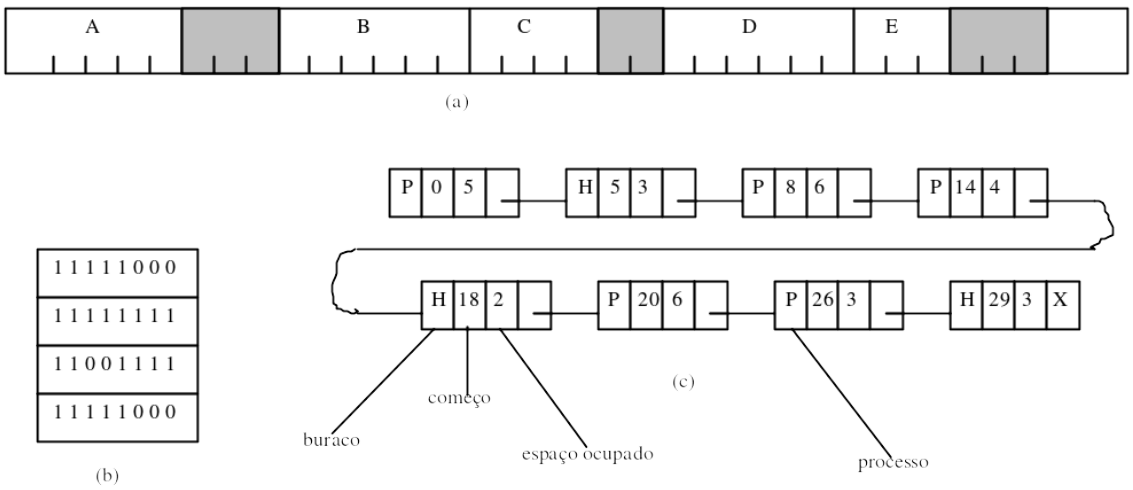
a. existe um buraco (região não ocupada) de memória próximo ao processo: neste caso, basta alocar memória desse buraco;
b. o processo está cercado por outros processos, ou o buraco que existe não é suficiente: neste caso, podemos tomar alguma das seguintes ações:
    b.1. mover o processo para um buraco de memória maior (grande o suficiente para as novas exigências do processo);
    b.2. se não houver tal espaço, alguns processos devem ser retirados da memória, para deixar espaço para esse processo;
    b.3. se não houver espaço no disco para outros processos, o processo que pediu mais espaço na memória deve ser morto (*kill*).

Quando se espera que diversos processos cresçam durante a execução, pode ser uma boa idéia reservar espaço extra para esses processos quando os mesmos são criados, para eliminar a sobrecarga de lidar com movimentação ou troca de processos.

Trataremos agora das três formas principais de cuidar da utilização de memória: mapas de bits, listas e sistemas de desabrochamento.

4.2.2. Gerenciamento de memória com mapa de bits

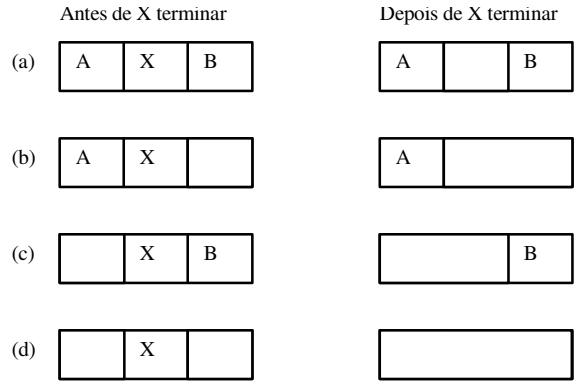
Este método consiste em formar um mapa de bits. A memória é subdividida em unidades de um certo tamanho. A cada uma dessas unidades é associado um bit que, se for 0 indica que essa parte da memória está livre, e se for 1, indica que ela está ocupada. O tamanho da unidade que é associada com um bit deve ser cuidadosamente escolhido, entretanto, mesmo com a associação de apenas 4 bytes de memória para cada bit no mapa, a parte da memória gasta com esse mapa é de apenas 3%. Se escolhemos mais de 4 bytes, o espaço ocupado pela tabela será menor, mas o desperdício de memória crescerá. A figura abaixo mostra um trecho de memória (a) e o mapa de bits associado (b).



A desvantagem desse método, é que, quando um novo processo que ocupa **k** unidades de memória deve ser carregado na memória, o gerenciador deve percorrer o mapa de bits para encontrar **k** bits iguais a zero consecutivos, o que não é um processo simples.

4.2.3. Gerenciamento de memória com listas ligadas

Neste caso, mantemos uma lista ligada de segmentos alocados e livres, sendo que cada segmento é um processo ou um buraco entre dois processos. Na figura anteriormente apresentada, temos também a lista associada ao trecho de memória indicado (c). H indica um buraco e P um processo. A lista apresenta-se em ordem de endereços, de forma que quando um processo termina ou é enviado para o disco, a atualização da lista é simples: cada processo, desde que não seja nem o primeiro nem o último da lista, apresenta-se cercado por dois segmentos, que podem ser buracos ou outros processos, o que nos dá as quatro possibilidades mostradas na figura abaixo:



Note que os buracos adjacentes devem ser combinados num único.

Vejamos agora alguns algoritmos que podem ser utilizados para escolher o ponto em que deve ser carregado um processo recém-criado ou que veio do disco por uma troca (foi *swapped in*). Assumimos que o gerenciador de memória sabe quanto espaço alocar ao processo:

first fit (primeiro encaixe): neste caso, o algoritmo consiste em percorrer a fila até encontrar o primeiro espaço em que caiba o processo. É um algoritmo rápido;

next fit (próximo encaixe): o mesmo que o algoritmo anterior, só que ao invés de procurar sempre a partir do início da lista, procura a partir do último ponto em que encontrou. Simulações mostraram que esse algoritmo apresenta um desempenho ligeiramente melhor que o anterior

best fit (melhor encaixe): consiste em verificar toda a lista, e procurar o buraco que tiver espaço mais próximo das necessidades do processo. É um algoritmo mais lento e, além disso, simulações demonstram que tem um desempenho pior que o *first fit*, devido ao fato de tender a encher a memória com pequenos buracos sem nenhuma utilidade;

worst fit (pior encaixe): sugerido pelo fracasso do algoritmo anterior, consiste em pegar sempre o maior buraco disponível. Simulações mostraram que também seu desempenho é ruim.

Os quatro algoritmos podem ter sua velocidade aumentada pela manutenção de duas listas separadas, uma para processos e outra para buracos.

Ainda outro algoritmo possível, quando temos duas listas separadas é o *quick fit* (encaixe rápido), que consiste em manter listas separadas para alguns dos tamanhos mais comuns especificados (por exemplo, uma fila para 2k, outra para 4k, outra para 8k, etc.). Neste caso, a busca de um buraco com o tamanho requerido é extremamente rápido, entretanto, quando um processo termina, a liberação de seu espaço é complicada, devido à necessidade de reagrupar os buracos e modificá-los de fila.

4.2.4. Gerenciamento de memória com sistema de desabrochamento

Este método se utiliza do fato de que o espaço de endereçamento da maioria dos sistemas é uma potência de dois. Para verificar o funcionamento desse algoritmo, observe a figura seguinte, onde são realizados pedidos sucessivos de 70k, 35k, 80k, liberação dos 70k inicial e outro pedido de 60k.

							Buracos
Inicial	1024						1
Pede 70	A	128		256		512	3
Pede 35	A	B	64	256		512	3
Pede 80	A	B	64	C	128	512	3
Retorna A	128	B	64	C	128	512	4
Pede 60	128	B	D	C	128	512	3
Retorna B	128	64	D	C	128	512	4
Retorna D	256			C	128	512	3
Retorna C	1024						1

Para entender o processo devemos considerar que:

a. o espaço é sempre alocado em unidades de potência de dois

b. é mantida uma lista para cada valor de potência de dois, desde um até o máximo que cabe na memória

c. quando um pedido de dada quantidade de memória (já arredondado para potência de dois) é feito, se necessário blocos maiores vão sendo sucessivamente divididos em dois até fornecer um bloco do tamanho requisitado

d. cada grupo de dois blocos que provieram da divisão de um bloco maior forma um botão. Sempre que os dois blocos são novamente liberados, o bloco maior é novamente formado

(p.ex., quando dois blocos de 64k que provieram de um mesmo bloco de 128k ficam novamente livres, eles se juntam novamente em um bloco de 128k)

Note que, quando um bloco é liberado, o algoritmo apenas precisa verificar na lista correspondente ao tamanho do mesmo se está livre o outro bloco com o qual este se pode combinar para formar um bloco maior.

O grave problema desse algoritmo é a ineficiência na utilização da memória: Quando um bloco de 35k é pedido, é alocado um bloco de 64k, e os 29k restantes são desperdiçados. Isto é chamado de fragmentação interna, pelo fato do espaço desperdiçado ser interno ao bloco. Chamamos de fragmentação externa quando espaços são desperdiçados entre os blocos.

4.2.5. Alocação de espaço de troca (swap)

Chamamos de espaço de troca ao espaço ocupado no disco pelos processos que aí estão guardados pelo fato de que foram retirados da memória devido a uma troca (*swap*).

Os algoritmos para gerenciar o espaço alocado em disco para *swap* são os mesmos apresentados para o gerenciamento da memória, com a diferença de que em alguns sistemas, cada processo tem no disco um espaço reservado para o mesmo, de forma que não está, como na memória, sendo constantemente mudado de lugar.

Um fator adicional de diferença é o fato de que, pelos discos serem dispositivos de bloco, a quantidade de espaço reservado para os processos no disco deverem ser um múltiplo do tamanho do bloco.

4.3. Memória virtual

Quando os programas começaram a ficar grandes demais para a quantidade de memória necessária, a primeira solução adotada foi a utilização de *overlay*. Nesta técnica, o programa era subdividido em partes menores, chamadas *overlays*, e que podiam ser rodadas separadamente. Quando um dos *overlays* terminava a execução, um outro poderia ser carregado na mesma posição de memória utilizada pelo anterior (isto existe ainda em compiladores como o Turbo-Pascal).

O problema com este método é que todo o trabalho de divisão de programas em *overlays*, que aliás não é simples, deve ser realizado pelo programador.

A técnica de memória virtual é uma outra forma de executar um programa que não cabe na memória existente, mas não apresenta os inconvenientes dos *overlays*, por ser realizada de forma automática pelo próprio computador. Neste caso, partes do programa, dos dados, e da pilha são mantidas no disco, sendo que existe uma forma de decisão cuidadosa de quais devem permanecer no disco e quais na memória. Da mesma forma, podemos alocar diversos processos na memória virtual, de forma que cada um pensa ter uma quantidades de memória que somadas ultrapassam a quantidade real de memória.

4.3.1. Paginação

Paginação é uma técnica muito utilizada em sistemas com memória virtual. Antes, estabeleçamos o conceito de espaço virtual.

Chamamos de **espaço virtual** ao espaço de memória que pode ser referenciado por um programa qualquer em dado processador. Por exemplo, um processador com endereçamento de 16 bits possui um espaço virtual de 64k bytes (se o endereçamento for em bytes). Quando uma instrução como:

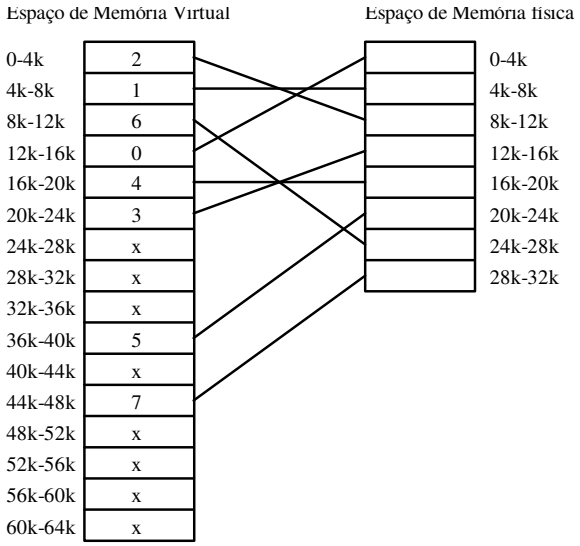
```
LD  A,(1000h)  ; carrega acumulador com o conteúdo do endereço 1000 (hexa)
```

é apresentada no processador Z80, o 1000h corresponde a um endereço **virtual**, de um espaço de endereçamento virtual de 64k bytes.

Em um computador **sem** memória virtual, o endereço virtual corresponde ao endereço efetivamente colocado no duto de endereçamento da memória. No exemplo acima, seria colocado no duto de endereços o valor binário correspondente a 1000h.

Quando o computador possui memória virtual, esse endereço virtual é enviado para uma **unidade de gerenciamento de memória**, MMU (*memory management unit*), que corresponde a um chip ou um módulo da UCP que translada esse endereço virtual em um endereço físico, de acordo com uma tabela.

A operação da MMU pode ser explicada conforme apresentado na figura abaixo:

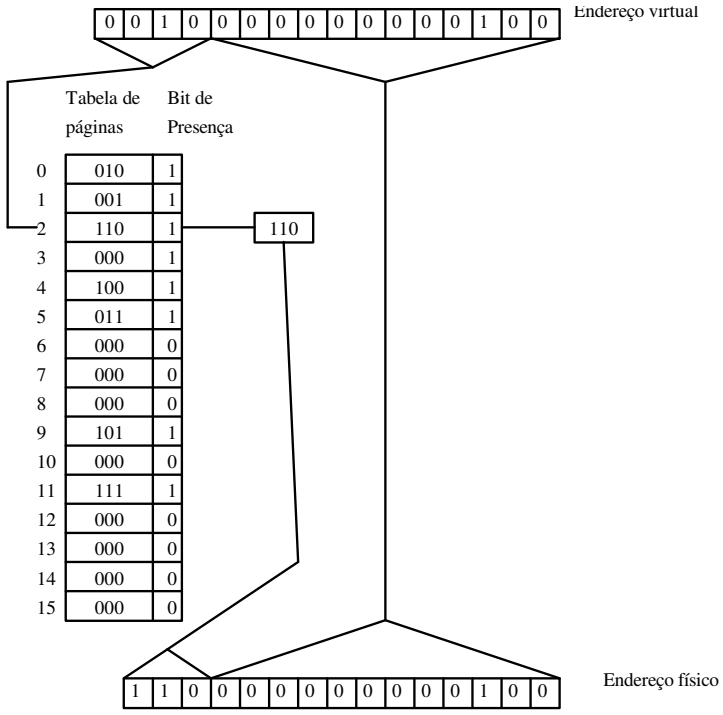


O espaço de endereços virtuais é dividido em unidades chamadas **páginas**, e o espaço de memória física é dividido em unidades chamadas **quadros de página**, de mesmo tamanho das páginas. A MMU tem uma tabela que indica, para cada página, qual o quadro de página que corresponde à mesma. No exemplo da figura, temos um espaço virtual de 64k bytes, uma memória física de 32k bytes, e páginas de 4k bytes. Se o processador tenta acessar o endereço 0, a MMU verifica que isto corresponde ao primeiro endereço da primeira página, verifica então que essa primeira página está alocada no terceiro quadro de página (i.e. o de número 2). Converte então esse endereço para 8192 (decimal) e envia esse endereço convertido para a memória. Note que nem o processador nem a memória precisaram ficar sabendo da existência de paginação.

No entanto, como nem todas as páginas do espaço virtual podem estar residentes na memória simultaneamente, ocorrerá o caso de que um acesso seja realizado para uma página que não está na memória. Para saber isto, a MMU mantém na tabela de translação um bit para cada página que indica se a mesma está presente na memória ou não. Se um acesso for realizado a uma página ausente, é gerada uma **falta de página** (*page fault*), o que chama uma rotina de tratamento de interrupção específica do S.O., que então se encarrega do carregamento da página faltante e o ajuste correspondente na tabela de translação.

A forma mais comum de implementação da MMU, é escolher alguns dos bits mais significativos do endereço virtual como indicadores do número de página, e o restante dos bits como um deslocamento

dentro dessa página. Por exemplo, na figura apresentada acima, de 16 bits do endereço virtual, 12 serão utilizados para o deslocamento (pois são necessários 12 bits para endereçar os 4k bytes de uma página), sendo os 4 restantes utilizados como um índice para qual das 16 páginas está sendo referenciada. A MMU portanto, pega os 4 bits do índice de página, acessa a posição correspondente da tabela de translação, verifica se a página está presente na memória, se não estiver, gera uma interrupção para carregamento, e depois verifica o valor colocado nessa entrada da tabela de translação e os junta aos 12 bits de deslocamento dentro da página. A figura abaixo mostra a operação da MMU.

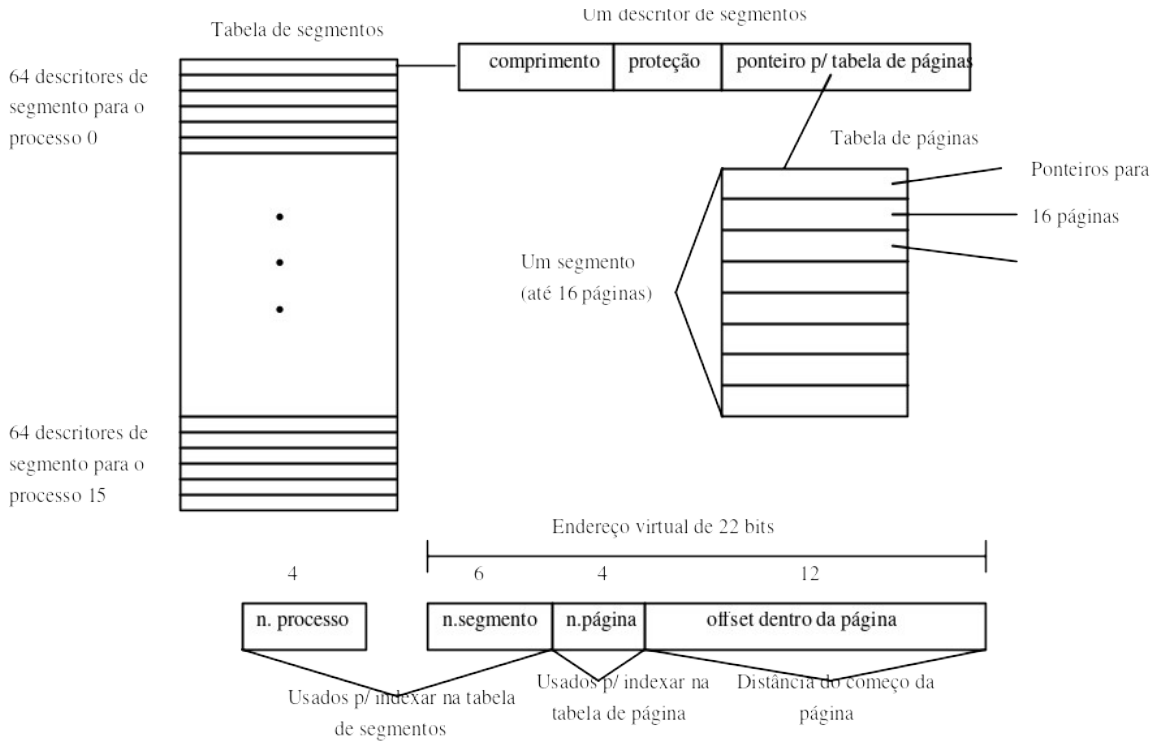


4.3.2. Segmentação

A paginação provê uma forma de se conseguir grandes espaços de endereçamento lineares em uma quantidade finita de memória física. Em algumas aplicações, no entanto, é preferível ter-se um espaço bidimensional. O espaço é bidimensional no sentido de que é dividido em um certo número de segmentos, cada um com dado número de bytes. Dessa forma, um endereçamento é sempre expresso da forma **(segmento, deslocamento)**. Os diferentes segmentos são associados a diversos programas ou mesmo arquivos, de forma que neste caso, os arquivos podem ser acessados como se fossem posições de memória. O sistema MULTICS foi o primeiro exemplo importante de utilização desse conceito.

No MULTICS, cada segmento estava relacionado com uma entidade lógica, como arquivos, rotinas, vetores. Veja que, diferentemente da paginação, na segmentação os programadores (ou os compiladores) levavam cuidadosamente em conta a segmentação, tentando colocar entidades diferentes em segmentos diferentes. Essa estratégia facilitou o compartilhamento de objetos entre processos diferentes.

A figura abaixo mostra uma implementação comum de segmentação em microcomputadores com o processador 68000.



Note que o *hardware* suporta até 16 processos, cada um com 1024 páginas de 4k bytes cada (isto é, cada processo com um endereço virtual de 4M bytes). A cada um dos 16 processos, a MMU associa uma seção com 64 descritores de segmento, sendo que cada descritor contém até 16 páginas. O descritor do segmento contém o tamanho do segmento (1 a 16 páginas), bits que indicam a proteção associada com o segmento, e um ponteiro para uma tabela de páginas.

Note também que para trocar de processo, a única coisa que o S.O. precisa fazer é alterar um registrador de 4 bits, e isso automaticamente mudará os acessos às tabelas.

Deve-se enfatizar que este é apenas um exemplo de esquema de segmentação, e não o único possível. O esquema utilizado no MULTICS é diferente.

4.4. Algoritmos de mudança de página

No momento em que é detectada a falta de uma página (i.e., ocorre um ***page fault***), o SO deve escolher uma das páginas atualmente residentes na memória para ser removida, de forma a liberar um quadro de página para a nova página. Deve-se tomar o cuidado, caso a página removida tenha sido alterada, de reescrevê-la no disco.

O problema que surge aqui é o seguinte: de que forma escolher a página a remover? É claro que devemos procurar remover uma página que não seja muito necessária aos processos em execução, isto é, devemos remover uma página que não será muito utilizada. Se removêssemos uma página amplamente utilizada, em pouco tempo uma referência à mesma seria realizada, o que ocasionaria um *page fault* e a necessidade de recarregar essa página, além de escolher outra para ser removida.

O que deve ser considerado então é: como determinar qual página não será muito utilizada? Os próximos ítens discutirão alguns métodos de se realizar essa escolha.

4.4.1. Mudança ótima de página

O método ótimo de mudança de página consiste no seguinte:

- Determinar, para cada página presente na memória, quantas instruções faltam para a mesma ser referenciada novamente (i.e., quantas instruções serão executadas antes que a página seja referenciada).
- Retirar a página que for demorar mais tempo para ser novamente referenciada.

Como é fácil perceber, o método delineado acima não pode ser implementado numa situação real, pois exigiria conhecimento de situações futuras (de que forma um SO conseguiria determinar quantas instruções faltam para um página ser referenciada?). Este método é utilizado apenas em simulações, onde temos um controle completo das execuções dos processos, para fins de comparação de algoritmos.

4.4.2. Mudança de página não recentemente utilizada (NRU)

Este método seleciona, para ser retirada, uma página que não tenha sido recentemente utilizada. Para determinar se uma página foi ou não recentemente utilizada, conta-se com o auxílio de 2 bits associados com cada uma das páginas na memória:

R: indica se a página foi referenciada

M: indica se a página foi modificada

Estes bits, presentes em muitos *hardwares* de paginação, são alterados automaticamente (por *hardware*) sempre que uma referência à página é realizada. Assim, quando uma instrução lê um dado em uma certa posição de memória, o bit R da página correspondente é colocado em 1; quando uma instrução escreve em uma dada posição de memória, os bits R e M da página correspondente são colocados em 1. Esses bits são colocados inicialmente em 0 quando uma página é carregada da memória. Uma vez que esses bits sejam colocados em 1, eles somente poderão voltar a 0 pelo carregamento de uma nova página ou por instruções específicas do SO.

O método para determinar se uma página foi recentemente utilizada é então o seguinte:

1. A cada tique do relógio, o SO coloca todos os bits R das páginas em 0. Quando ocorre um *page fault*, temos quatro categorias de páginas:
   classe 0: R=0, M=0
   classe 1: R=0, M=1
   classe 2: R=1, M=0
   classe 3: R=1, M=1
   A classe 0 corresponde a páginas que não foram nem alteradas nem referenciadas desde o último tique do relógio. A classe 2 corresponde às páginas que foram referenciadas pelo menos uma vez desde o último tique do relógio, sem haverem sido modificadas. A classe 3 corresponde a páginas que foram referenciadas e alteradas desde o último tique. A classe 1 corresponde a páginas que, apesar de já haverem sido modificadas desde que foram carregadas, não foram referenciadas desde o último tique.
2. Escolhe-se para retirar uma das páginas que pertencer à classe mais baixa (de número mais baixo) não vazia, no momento da ocorrência do *page fault*.

Esse algoritmo apresenta as vantagens de ser implementável de forma simples e eficiente.

Quando o *hardware* não dispõe dos bits R e M, o SO pode simular esses bits através da utilização de mecanismos de proteção de páginas, da seguinte forma:

1. Inicialmente, marcam-se todas as páginas como ausentes.
2. Quando uma página é referenciada, é gerado um *page fault*, pois a página está marcada como ausente.

3. O SO marca então essa página como presente, mas permite apenas leitura. Simultaneamente, o SO marca numa tabela interna o bit R simulado dessa página em 1.
4. Quando uma escrita for tentada nessa página, uma interrupção de acesso inválido será gerada, e o SO poderá então marcar a página como de leitura e escrita, e o bit M simulado correspondente em 1.

4.4.3. Mudança de página "primeira a entrar, primeira a sair" (FIFO)

Neste caso mantém-se uma fila de páginas referenciadas. Ao entrar uma nova página, ela entra no fim da fila, substituindo a que estava colocada no início da fila.

O problema com esse algoritmo é que pode retirar páginas muito valiosas (i. é., páginas que, apesar de estarem há muito tempo na memória estão sendo amplamente utilizadas).

Uma possível solução para esse problema consiste na utilização dos bits R e M:

1. Verificam-se os bits R e M da página mais antiga.
2. Se essa página for da classe 0, ela é escolhida para ser retirada.
3. Se não for, continua procurando na fila.
4. Se em toda a fila não é encontrada nenhuma página da classe 0, prossegue para as classes seguintes.

Uma variação dessa solução é o algoritmo conhecido como segunda chance:

1. Verifica o bit R da página mais velha (primeira da fila).
2. Se R for 0, utiliza essa página.
3. Se R for 1, põe R em 0 e coloca a página no fim da fila.
4. Prossegue analisando a fila, até encontrar uma página com R=0 (no pior caso será a primeira página, que foi colocada no final da fila)

Anomalia de Belady

Consiste numa anomalia presente no algoritmo de troca de página por fila, e corresponde a que, para algumas seqüências de pedidos de páginas, um aumento do número de quadros de página na memória, o número de *page faults* gerados aumenta, ao invés de diminuir como é o caso geral. Na figura abaixo temos um exemplo, com a seguinte seqüência de pedidos de página:

0 1 2 3 0 1 4 0 1 2 3 4

e memória com respectivamente 3 e 4 quadros de página. Note que para 3 quadros de página temos 9 *page faults*, enquanto que para 4 quadros temos 10 *page faults*.

	0	1	2	3	0	1	4	0	1	2	3	4
Página mais jovem	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Página mais antiga			0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P			P	P	

	0	1	2	3	0	1	4	0	1	2	3	4
Página mais jovem	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Página mais antiga				0	0	0	1	2	3	4	0	1
	P	P	P	P			P	P	P	P	P	P

Obs: Todas as páginas inicialmente

4.4.4. Mudança de página menos recentemente utilizada (least recently used, LRU)

Este método se baseia nas seguintes observações:

- Páginas muito utilizadas nas instruções mais recentes provavelmente permanecerão muito utilizadas nas próximas instruções.
- Páginas que não são utilizadas há tempo provavelmente permanecerão não utilizadas por muito tempo.

O algoritmo então consiste no seguinte: quando ocorre um *page fault*, retira-se a página que há mais tempo não é referenciada.

O problema básico com esse algoritmo é que sua implementação é muito dispendiosa: deve-se manter uma lista com todas as páginas na memória, com as mais recentemente utilizadas no começo e as menos recentemente utilizadas no final, e esta lista deve ser alterada a cada referência de memória. Note-se que a implementação por *software* é completamente inviável. Vejamos agora duas possíveis soluções por *hardware*.

Na primeira, mantemos no processador um contador C, com um número relativamente grande de bits (p.ex.: 64 bits). Com esse contador realizamos o seguinte procedimento:

1. A cada instrução executada incrementamos esse contador.
2. Cada entrada na tabela de páginas tem associado um campo com tantos bits quanto os do contador C.
3. Depois de cada referência à memória, o contador C é incrementado, e seu valor armazenado na entrada correspondente da tabela de páginas.
4. Quando ocorre um *page fault*, escolhemos a página que apresenta menor valor no campo correspondente ao contador (pois essa será a página que há mais tempo foi referenciada pela última vez).

Na segunda solução, para n quadros de páginas, temos uma matriz $n \times n$ de bits, e agimos da seguinte forma:

1. Quando a página i é referenciada, colocamos em 1 todos os bits da linha i da tabela, e em seguida colocamos em 0 todos os bits da coluna i.
2. Quando ocorre um *page fault*, retiramos a página que apresentar o menor número na linha correspondente (este número é interpretado como a associação de todos os bits da linha, da esquerda para a direita).

4.4.5. Simulando LRU em software

Nas implementações de LRU apresentadas até o momento, consideramos sempre existir *hardware* especial. Quando não existe *hardware* especial para sua implementação, podemos realizar uma aproximação de LRU por *software*, chamada NFU (não freqüentemente utilizada):

1. Um contador é mantido, em *software*, para cada página.
2. A cada interrupção de relógio, o bit R correspondente a cada uma das páginas é somado a esse contador (portanto, se a página foi referenciada dentro do último intervalo de relógio, seu contador é incrementado, se ela não foi referenciada, o contador permanece no mesmo valor).
3. Quando ocorre um *page fault*, escolhe-se a página com menor valor nesse contador.

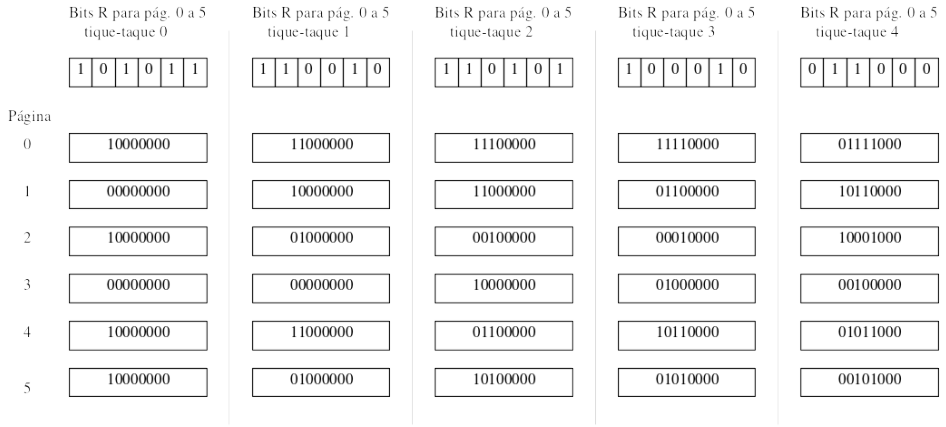
O grande problema com esse algoritmo é que se uma página é intensamente utilizada durante um certo tempo, ela tenderá a permanecer na memória, mesmo quando não for mais necessária (pois adquiriu um valor alto no contador).

Felizmente, uma modificação simples pode ser efetuada no método, de forma a evitar esse inconveniente. A alteração consiste no seguinte:

1. Os contadores são deslocados um bit à direita antes de somar R.

2. R é somado ao bit mais à esquerda (mais significativo) do contador, ao invés de ao bit mais à direita (menos significativo).

Esse algoritmo é conhecido como algoritmo de ***aging***. É óbvio que uma página que não foi referenciada há 4 tiques terá 4 zeros à esquerda, e portanto um valor baixo, colocando-a como candidata à substituição. Veja a figura abaixo.



Este algoritmo tem duas diferenças principais com relação ao algoritmo de LRU:

1. Não consegue decidir qual a referência mais recente com intervalos menores do que um tique.
2. O número de bits do contador é finito e, portanto, quando este chega a zero, não conseguimos mais distinguir as páginas mais antigas. Por exemplo, com 8 bits no contador, uma página não referenciada há 9 intervalos não pode ser diferenciada de uma página não referenciada há 1000 intervalos. No entanto, em geral 8 bits são suficientes para determinar que uma página já não é mais necessária.

4.5. Considerações de projeto para sistemas de paginação

Veremos alguns pontos que devem ser considerados, além do algoritmo de paginação propriamente dito, para que um sistema de paginação tenha bom desempenho.

4.5.1. Modelo do conjunto ativo (working set)

Num sistema puro de paginação, que também pode ser chamado de sistema de **paginação por demanda**, o sistema começa sem nenhuma das páginas na memória, e elas vão sendo carregadas à medida que forem necessárias, seguindo o algoritmo de substituição de página escolhido. Esta estratégia pura pode ser melhorada, como veremos a seguir.

Um primeiro dado importante a considerar é a existência, na maioria dos processos, de uma **localidade de referências**, o que significa que os processos mantém, em cada uma das fases de sua execução, referências a frações pequenas do total do número de páginas necessárias a ele. Um exemplo disso é um compilador, que durante um certo tempo acessa as páginas correspondentes à análise sintática, depois essas páginas deixam de ser necessárias, passando-se então a utilizar as de análise semântica, e assim por diante.

Baseado nesse fato, surge o conceito de **conjunto ativo** (***working set***), que corresponde ao conjunto de páginas correntemente em uso por um dado processo. É fácil de notar que, se todo o conjunto ativo de um processo estiver na memória principal, ele executará praticamente sem gerar *page faults*, até que passe a uma nova fase de processamento. Por outro lado, se não houver espaço para todo o conjunto

ativo de um processo, este gerará muitos *page faults*, o que ocasionará uma grande piora em seu tempo de execução, devido à necessidade de constantes trocas de páginas entre memória e disco. Aqui surge o conceito de ***trashing***, que ocorre quando um processo produz um grande número de *page faults* durante a execução de um número relativamente pequeno de instruções.

Consideremos então a seguinte questão: o que fazer quando um processo novo é iniciado ou, num sistema com *swap*, quando um processo que estava no disco deve ser colocado na memória? Se deixarmos que esse processo gere tantos *page faults* quanto necessários para a carga de seu conjunto ativo, teremos uma execução muito lenta. Devemos então, de alguma forma, determinar qual o conjunto ativo do processo e colocá-lo na memória antes de permitir que o processo comece a executar. Este é o chamado **modelo do conjunto ativo** (***working set model***). O ato de carregamento adiantado das páginas (antes da ocorrência do *page fault*) é chamado **pré-paginação**.

Vejamos agora algumas considerações com relação aos tamanhos do conjunto ativo. Se a soma total dos conjuntos ativos de todos os processos presentes na memória for maior que a quantidade de memória disponível, então ocorrerá *trashing*. Portanto, devemos escolher os processos residentes em memória de forma que a soma de seus conjuntos ativos não seja maior do que a memória disponível.

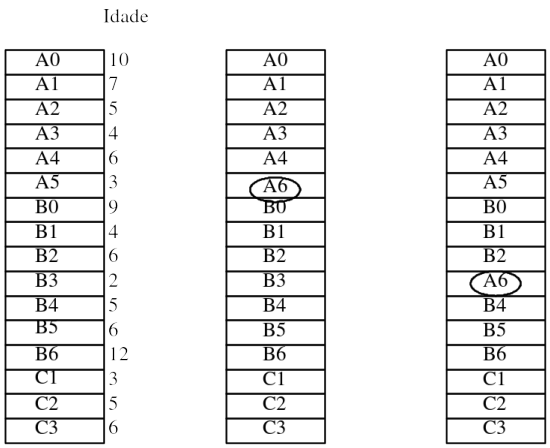
Devemos agora considerar a questão: como determinar quais as páginas de um processo que pertencem a seu conjunto ativo? Uma possível solução para isto é utilizar o algoritmo de *aging*, considerando como parte do conjunto ativo apenas as páginas que apresentam ao menos um bit em 1 em seus n primeiros bits (i.é., qualquer página que não seja referenciada por n intervalos consecutivos do relógio é retirada do conjunto ativo do processo). O valor de n deve ser definido experimentalmente.

4.5.2. Rotinas de alocação local × global

Devemos agora determinar de que forma a memória será alocada entre os diversos processos executáveis que competem por ela.

As duas estratégias básicas são chamadas de **alocação local** e **alocação global**. Na alocação local, quando é gerado um *page fault* em certo processo, retiramos uma das páginas do próprio processo para a colocação da nova página. Na alocação global, não levamos em consideração a qual processo a página a ser retirada pertence, isto é, será retirada a página que for determinada pelo algoritmo de paginação, independente dela pertence ao processo que gerou o *page fault* ou não.

Para exemplificar a diferença entre as duas técnicas, veja a figura abaixo onde, na situação da figura (a) é gerado um *page fault* para a página A6. Se seguirmos uma estratégia local, a página retirada será a A5, enquanto que para uma estratégia global retiraremos a página B3. (Os números ao lado das páginas na figura indicam o valor do contador do algoritmo de *aging*).



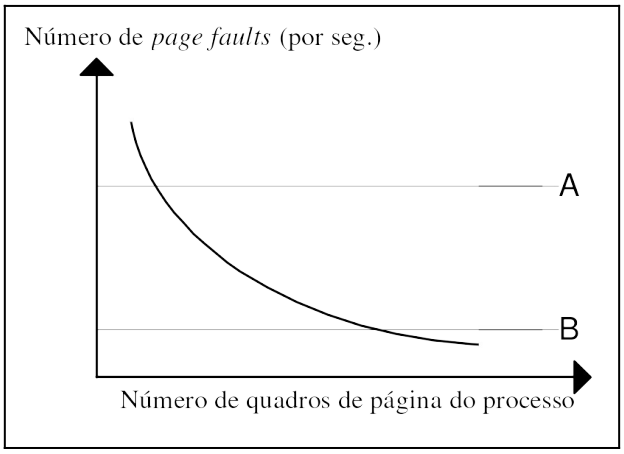
É fácil de notar que a estratégia local mantém fixa a quantidade de memória destinada a cada processo, enquanto que a estratégia global muda dinamicamente a alocação de memória. De modo geral, os algoritmos globais apresentam melhor eficiência, devido aos seguintes fatores:

- Se o conjunto ativo de um processo aumenta, a estratégia local tende a gerar *trashing*.
- Se o conjunto ativo de um processo diminui, a estratégia local tende a desperdiçar memória.

Entretanto, se nos decidimos por utilização de uma estratégia global, devemos resolver a seguinte questão: de que forma o sistema pode decidir dinamicamente (i.é., a todo instante), quantos quadros de página alocar para cada processo (em outras palavras, qual o tamanho do *working set* do processo a cada instante)?

Uma possível solução é realizar o monitoramento pelos bits de *aging*, como indicado acima. Entretanto, isto apresenta um problema, que é o de que o conjunto ativo de um processo pode mudar em poucos microssegundos, enquanto que as indicações de *aging* só mudam a cada tique do relógio (digamos, cada 20ms).

Uma solução melhor para esse problema é a utilização de um algoritmo de alocação por **freqüência de falta de página** (PFF: *page fault frequency*). Esta técnica é baseada no fato de que, com exceção da anomalia de Belady (que ocorre apenas para o algoritmo de fila), a taxa de falta de página tende a decrescer com o aumento do número de quadros de página alocados ao processo. Isto poderia ser ilustrado num gráfico como o abaixo (esquemático).



Os níveis A e B indicados no gráfico têm o seguinte significado: o nível A representa o nível máximo admissível de faltas de página para um processo, enquanto o nível B representa o nível mínimo admissível de falta de páginas.

Se um processo apresenta uma taxa de falta de páginas maior que o nível A, então ele estará trabalhando com uma quantidade de páginas muito baixa, insuficiente para seu conjunto ativo, e portanto devem ser alocadas mais páginas para esse processo. Por outro lado, se um processo está trabalhando com uma taxa de falta de páginas menor que o nível B, então isto significa que ele está ocupando mais memória do que é conveniente para o sistema como um todo, pois essa memória poderia ser útil para outro processo que esteja rodando com taxas maiores que A. Portanto, o algoritmo consiste em:

1. Anotar, para cada processo, sua taxa de geração de *page faults*.
2. Se essa taxa estiver acima de A, aumentar o número de quadros para esse processo.
3. Se a taxa estiver abaixo de B, diminuir o número de quadros do processo.
4. Se não existe memória suficiente para manter todos os processos operando abaixo de A, alguns processos devem ser enviados para o disco (*swap*).

4.5.3. Tamanho de página

Um outro fator que deve ser cuidadosamente estudado é o tamanho da página. Existem considerações em favor tanto de páginas pequenas como de páginas grandes, como veremos. Favoráveis a páginas pequenas são:

1. Na média, metade da página final de qualquer segmento é desperdiçada (pois os programas e áreas de dados não terminam necessariamente em múltiplos do tamanho de página). Isto é conhecido como fragmentação interna.
2. Um programa que consista em processos pequenos pode executar utilizando menos memória quando o tamanho da página é pequeno.

Favoravelmente a páginas grandes podemos citar:

1. Quanto menor o tamanho da página, maior o tamanho da tabela de páginas (para uma dada quantidade de memória virtual).
2. Quando uma página precisa ser carregada na memória, temos um acesso ao disco, com os correspondentes tempos de busca e espera de setor, além do de transferência. Assim, se já transferimos uma quantidade maior de bytes a cada vez, diminuímos a influência dos dois primeiros fatores, aumentando a eficiência.
3. Se o processador central precisa alterar registradores internos referentes à tabela de páginas a cada chaveamento de processos, então páginas maiores necessitarão menos registradores, o que significa menor tempo de chaveamento de processos.

Tamanhos de página comumente utilizados são: 512, 1024, 2048 e 4096 bytes.

4.5.4. Considerações de implementação

Normalmente, quando queremos implementar um sistema de paginação, devemos considerar alguns fatos, que complicam um pouco seu gerenciamento, e dos quais apresentaremos alguns exemplos a seguir.

Volta de instruções

O problema aqui é o seguinte: quando um *page fault* é gerado durante uma instrução de múltiplos bytes, como fazemos para continuar a execução da instrução após o carregamento da página? Por exemplo, suponha que um computador acabou de ler o código de uma operação a executar no endereço 1023 e, ao tentar ler o endereço do operando (que seria encontrado, por exemplo, nos endereços 1024 e 1025), um *page fault* é gerado. De que forma o SO pode saber que a instrução que estava sendo executada era a do endereço 1023, de forma a reinicializá-la? Ou, ainda pior, se a instrução era de auto-incremento ou auto-decremento de algum registrador, de que forma o SO pode saber se o registrador já foi incrementado ou decrementado? (É claro que ele precisa dessa informação, pois necessita retomar a instrução que foi interrompida, e se o registrador já havia sido alterado, ele não deve ser alterado novamente.)

Uma solução para esses problemas, adotada pelo processador 68010, é a de manter internamente as seguintes informações:

- Endereço da instrução que está sendo executada (não importa que o contador de programa já tenha sido incrementado para buscar as informações sobre o operando).
- Indicação de todos os registradores que já foram auto-incrementados ou auto-decrementados.

Essas informações seriam acessíveis ao SO, que poderia então utilizá-las para o restauramento do estado do processador antes do início da execução da instrução que foi interrompida pelo *page fault*.

Trancamento de páginas na memória

Quando um processo bloqueia aguardando a transferência de dados de um dispositivo para um *buffer*, um outro processo é selecionado para execução. Se este outro processo gera um *page fault*, existe a possibilidade de que a página escolhida para ser retirada seja justamente aquela onde se encontra o *buffer* que estava aguardando a transferência de dados do dispositivo. Se o dispositivo tentar então transferir para esse *buffer*, ele pode estragar a nova informação carregada nessa página. Existem basicamente duas soluções para esse problema:

1. Trancamento da página que contém o *buffer*, impedindo que a mesma seja retirada da memória.
2. Realizar as transferências de dispositivos sempre para *buffers* internos do *kernel* e, após terminada a transferência, destes para o *buffer* do processo.

Páginas compartilhadas

Um outro problema que surge está relacionado com o fato de que diversos processos podem estar, em um dado momento, utilizando o mesmo programa (por exemplo, diversos usuários rodando um editor, ou um compilador). Para evitar duplicação de informações na memória, é conveniente que essas informações comuns sejam armazenadas em páginas que são compartilhadas pelos processos que as utilizam.

Um problema que surge logo de início é o seguinte: algumas partes não podem ser compartilhadas, como por exemplo, os textos que estão sendo editados. A solução para este problema é simples: basta saber quais as páginas que são apenas de leitura (por exemplo: código do programa editor), permitindo que estas sejam compartilhadas, e quais as que são de leitura e escrita (por exemplo o texto sendo editado), impedindo seu compartilhamento.

Um outro problema mais difícil de remover é o seguinte:

- Suponha que dois processos, A e B, estão compartilhando o código de um editor.
- Se A termina a edição, e portanto sai da memória, algumas de suas páginas (as que correspondem ao código do editor) não poderão ser retiradas, devido a serem compartilhadas, pois se isto ocorresse, seriam gerados muitos *page faults* para o processo B, o que representaria uma grande sobrecarga para o sistema.
- Se um dos processos é enviado para o disco (*swap*), temos uma situação semelhante ao caso do término de um deles.

A solução para isso é manter alguma forma de estrutura de dados que indique quais das páginas residentes estão sendo compartilhadas. Este processo, entretanto, não é simples.

Exercícios

4.1. Usando o modelo simplificado apresentado para multiprogramação (veja gráfico no item 4.1.1.), podemos prever o crescimento da eficiência de utilização da UCP com o crescimento do grau de multiprogramação. Suponha um computador com 2M bytes de memória, no qual o S.O. ocupa 512k bytes e cada processo de usuário também ocupa 512k bytes. Se todos os processos permanecem 60% do tempo em espera de E/S, em quanto o desempenho crescerá se for adicionado mais 1M de memória?

4.2. Considere um sistema de troca (*swapping*) no qual a memória consista nos seguintes tamanhos de buraco, em ordem: 10k, 4k, 20k, 18k, 7k, 9k, 12k, 15k. Qual buraco é dado a cada um dos seguintes pedidos de memória sucessivos:
a) 12k

b) 10k
c) 9k
para o algoritmo de *first fit*? E para *best fit*, *worst fit* e *next fit*?

4.3. Um minicomputador usa o sistema de desabrochamento para gerenciamento de memória. Inicialmente ele possui 1 bloco de 256k no endereço 0. Depois da chegada de pedidos sucessivos de 5k, 25k, 35k e 20k, quantos blocos ainda restam e qual seus tamanhos e endereços?

4.4. Se uma instrução toma 1μs e uma falta de página (*page fault*) toma mais *n* μs, dê uma fórmula para o tempo de instrução efetivo, se ocorre uma falta de página a cada *k* instruções, em média.

4.5. Utilizando a tabela de páginas apresentada na figura do item 4.3.1., indique qual o endereço físico que corresponde aos seguintes endereços virtuais (em decimal):
a) 20
b) 4100
c) 8300

4.6. Imagine um computador que usa MMU segmentada, como a mostrada na figura do item 4.3.2., exceto que o número do processo tem apenas 3 bits, e o espaço de endereçamento virtual é de 8M, com páginas de 4k e segmentos de 32k. Quantas palavras são necessárias para manter todos os ponteiros de páginas? Se o tamanho do segmento for mudado para 64k, sem mudar nenhum dos outros parâmetros (i.e., menos segmentos, porém maiores), como isto afetará o número de ponteiros para tabela de páginas necessários?

4.7. Um computador tem 4 quadros de páginas. O tempo de carga, o tempo de último acesso e os bits R e M são apresentados no quadro abaixo (todos em unidades do relógio).

Página	Carga	Última referência	R	M
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

Qual será a página retirada por cada um dos seguintes algoritmos:
(a) NRU
(b) FIFO
(c) segunda chance
(d) LRU

4.8. Se utilizamos a troca de página por fila, em uma memória com 4 quadros de página e 8 páginas virtuais, quantos *page faults* ocorrerão com as referências de páginas na ordem abaixo:
0 1 7 2 3 2 7 1 0 3
se os 4 quadros de páginas estão inicialmente vazios? Repita o problema para o algoritmo LRU.

4.9. Um computador provê, para cada processo, um endereçamento virtual de 65536 bytes, divididos em páginas de 4096 bytes. Um programa particular tem o tamanho do segmento de texto (código do programa) de 32768 bytes, um segmento de dados de 16386 bytes e um segmento de pilha de 15870 bytes. Ele caberá no espaço de endereçamento? E se as páginas fossem de 512 bytes? (Lembre-se que uma mesma página não pode conter dados de segmentos distintos.)

4.10. Foi observado que o número médio de instruções executadas entre dois *page faults* consecutivos é diretamente proporcional ao número de quadros de página alocados ao processo. Se a memória disponível é duplicada, o intervalo médio entre *page faults* é também

duplicado. Suponha que uma instrução normal tome 1μs, mas se um *page fault* ocorre ela toma 2001μs. Se um programa leva 60s para rodar, e nesse intervalo ele gera 15000 *page faults*, quanto tempo ele levará para rodar se dispuser do dobro de memória?

5. Sistema de Arquivos

5.1. O sistema de arquivos conforme visto pelo usuário

Para o usuário de um sistema operacional, o sistema de arquivos representa a parte com que ele tem mais contato. As questões principais de um sistema de arquivos, sob esse ponto de vista, são as seguintes:

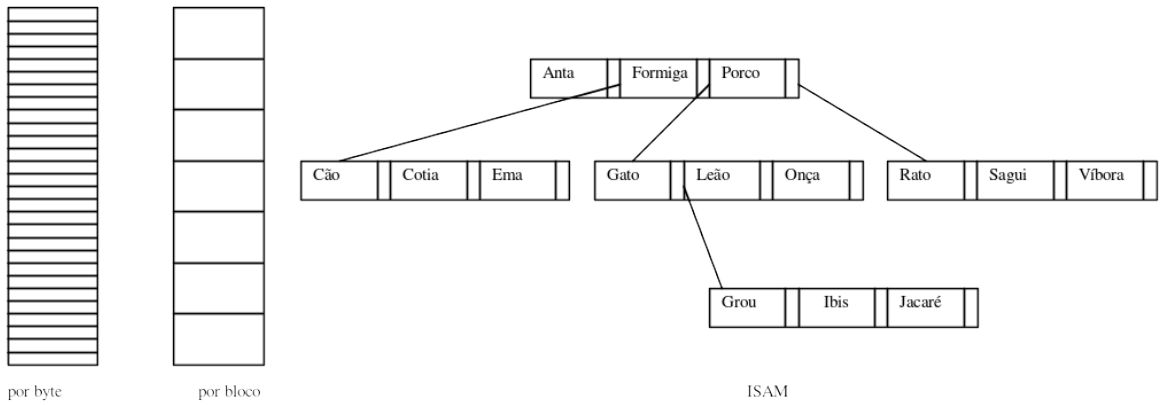
- O que é um arquivo?
- Como se dão nomes aos arquivos?
- Como se protegem arquivos contra acessos não autorizados?
- Que operações podem ser realizadas sobre um arquivo?

Para cada sistema operacional, as respostas a essas perguntas são dadas de forma diferente.

5.1.1. Pontos básicos com relação a arquivos

Os S.O. permitem que usuários definam objetos chamados **arquivos**, e apresentam operações especiais (realizadas por **chamadas de sistema**), que permitem agir sobre o mesmo (criar, destruir, escrever, ler, etc.).

Três formas comuns de organização de arquivos são apresentadas na figura abaixo:



No primeiro caso, os arquivos são considerados como constituídos de uma seqüência de bytes (é o caso do UNIX). No segundo caso, os arquivos são considerados como consistindo de uma seqüência de blocos, cada um de um dado tamanho (é o caso do CP/M). No terceiro caso, o arquivo é organizado em blocos, sendo que cada bloco consiste de certo número de regiões de dados e também de ponteiros para outros blocos semelhantes, formando-se uma árvore de blocos para cada arquivo (é o padrão ISAM: *indexed sequential access method*).

Um outro ponto importante com relação a arquivos é sua utilização para obter a independência de dispositivo, procurando os S.O. fornecer acessos aos arquivos independentemente dos dispositivos

utilizados para o armazenamento do arquivo. Em geral, os S.O. caracterizam os arquivos em alguns tipos. Por exemplo, o UNIX caracteriza os arquivos em: regulares, diretório, especiais (subdividido em de bloco e de caracter) e *pipes*. Nestes casos, algumas pequenas diferenças existem na utilização dos diversos tipos de arquivos.

Uma forma de diferenciar arquivos fornecida por alguns S.O. é a extensão (p.ex.: os .EXE, .COM, .BAT, etc. do MS-DOS). Com relação a isso, as atitudes dos S.O. para com as extensões variam desde alguns que não utilizam a extensão para nada, servindo a mesma apenas como mnemônico para o usuário, até sistemas em que o tipo de extensão é extremamente importante para o sistema (p.ex. ele poderia se recusar a rodar o compilador PASCAL em qualquer arquivo que não tivesse extensão .PAS).

5.1.2. Diretórios

Os diretórios são organizados como um conjunto de diversas entradas, uma para cada arquivo. Algumas possibilidades de organização dos diversos arquivos em diretórios são as seguintes:

- um único diretório para todos os arquivos (p.ex.: CP/M);
- um diretório para cada usuário;
- uma hierarquia de diretórios (p.ex.: MS-DOS e UNIX).

No último caso, surge a necessidade de se especificar a localização de um arquivo com relação à hierarquia de diretórios. O modo geralmente utilizado é a especificação de um **caminho** (*path*), que pode ser indicado de duas formas:

caminho absoluto: indica todo o caminho até o arquivo a partir da raiz da hierarquia

caminho relativo (associado a diretório de trabalho: *working directory*): indica como se faz para chegar até o arquivo partindo do diretório de trabalho atual.

5.2. Projeto de sistema de arquivos

No projeto de sistemas de arquivos, devemos estabelecer o seguinte:

- Como será feito o gerenciamento do espaço em disco?
- Como serão armazenados os arquivos?
- Como fazer tudo funcionar com eficiência e segurança?

Veremos agora, com alguns detalhes, formas de responder às duas primeiras perguntas.

5.2.1. Gerenciamento do espaço em disco

A forma de gerenciamento do espaço em disco é a principal consideração que deve ser feita pelo projetista de sistemas de arquivos.

Uma possibilidade seria, para cada arquivo, alocar espaço no disco correspondente ao número de bytes necessários, seqüencialmente (i.e., todos os bytes do arquivo serão consecutivos no disco). Surge um problema nesta solução quando há a necessidade de um arquivo crescer e não existe espaço consecutivo após o mesmo. Neste caso, seria necessário deslocar o arquivo para outra parte do disco, o que é lento e muitas vezes impossível (em discos cheios). Outro problema com este tipo de solução é a fragmentação do espaço em disco (da mesma forma que ocorria fragmentação da memória!).

A solução para este problema, então, é a alocação para cada arquivo de blocos de tamanho fixo, e uso de blocos não necessariamente contíguos no disco para um arquivo.

Escolhida esta opção, surge a seguinte questão: Qual o tamanho do bloco? A escolha de blocos de tamanho grande implica em que mesmo arquivos pequenos irão ocupar muito espaço no disco. Por outro lado, a escolha de blocos pequenos implica em um maior número de leituras em disco (para um dado tamanho de arquivo), o que implica em aumento das influências do tempo de busca de trilhas e espera de setor. Os tamanhos de blocos comumente encontrados são de 512 , 1K , 2K, 4K e 8K Bytes.

Resta-nos uma consideração importante: de que forma cuidar de quais blocos do disco estão livres e quais estão ocupados? Os métodos, como no caso de gerenciamento de memória, são basicamente dois:

- lista ligada
- *bit map* (mapa de bits)

A segunda opção é geralmente a melhor, principalmente quando todo o *bit map* pode ser mantido simultaneamente na memória. A primeira opção só é preferível quando o *bit map* não pode ser mantido todo na memória e o disco está bastante ocupado (isto pois, neste caso, com o método de *bit map*, se queremos encontrar espaço para um novo arquivo, devemos ficar carregando as diversas partes do *bit map* do disco para a memória, pois como o disco está cheio, é difícil de encontrar blocos vazios, o que torna o processo lento, enquanto que na lista ligada, basta carregar a lista de blocos vazios ao invés de ter que carregar as diversas partes do *bit map* uma por vez até achar o espaço necessário).

5.2.2. Armazenamento de arquivos

Vejamos agora de que forma podemos saber quais os blocos que estão alocados a um dado arquivo.

Um método possível é fazer uma lista ligada utilizando os próprios blocos, de forma que, num dado bloco (p.ex., de 1kbytes), alguns de seus bytes são utilizados para indicar qual o próximo bloco do arquivo.

Esta solução apresenta dois problemas:

- o número de bytes de um bloco não é potência de 2, o que é inconveniente em muitas aplicações
- para se buscar o *n*-ésimo bloco de um arquivo, devemos ler todos os *n*-1 blocos anteriores, além do próprio bloco, o que implica numa grande sobrecarga de leituras em disco quando pretendemos realizar acesso aleatório às informações do arquivo.

Um modo de utilizar a lista ligada sem os inconvenientes apresentados acima é o utilizado no MS-DOS, através de uma estrutura chamada FAT (*file allocation table*). Esta solução consiste em formar a lista dos blocos de um arquivo em uma estrutura especial, que pode ser carregada completamente na memória, eliminando simultaneamente os dois inconvenientes apontados. No MS-DOS, a FAT funciona da seguinte forma:

- a entrada no diretório correspondente a um arquivo indica o primeiro bloco ocupado por um arquivo
- a entrada na FAT correspondente a um bloco indica o próximo bloco alocado ao mesmo arquivo.

Os problemas com esta solução são os seguintes:

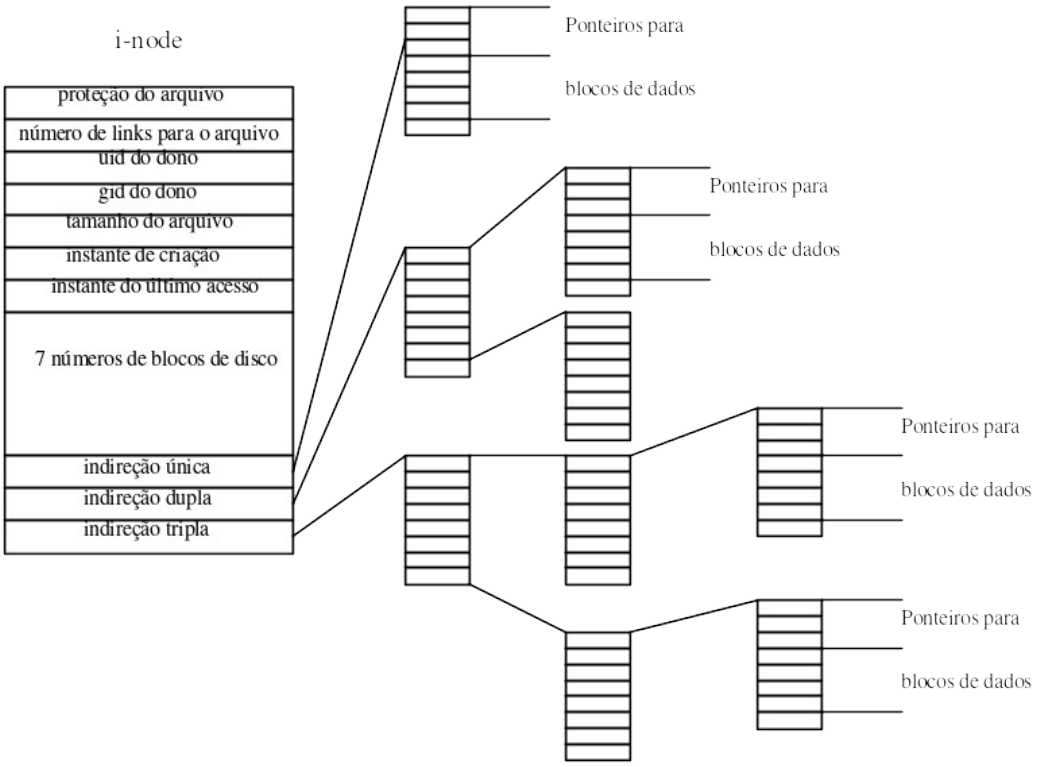
- como a FAT ocupa locais fixos no disco, o número de blocos que um disco pode possuir é limitado pelo número de blocos da FAT
- como cada entrada na FAT corresponde a um número fixo de bits (12 no caso de discos flexíveis), o número de blocos no disco é limitado (4096 no caso de disquete), sendo que se quisermos utilizar discos maiores, devemos alterar completamente a FAT.

No sistema UNIX, utiliza-se uma solução diferente, que apresenta grandes facilidades de expansão de tamanhos de dispositivos e arquivos. Podemos resumir da seguinte forma:

- as listas de blocos de cada arquivo são mantidas em locais diferentes (i.e., não existe um local fixo onde são guardadas as informações sobre os blocos de todos os arquivos)

- a cada arquivo, associa-se uma tabela, chamada *i-node*
- o *i-node* de um arquivo possui informações de proteção, contagem, blocos, entre outras
- no *i-node* existem 7 números de blocos diretos
- se o arquivo ocupa até 7 blocos, os números desses blocos são colocados nesse espaço reservado no *i-node*
- se o arquivo ocupa mais do que 7 blocos, um novo bloco é alocado ao arquivo, que será utilizado para armazenar tantos números de blocos desse arquivo quantos couberem no bloco. O endereço desse bloco será armazenado em um número de bloco de endereçamento indireto
- se ainda forem necessários mais blocos do que couberam nas estruturas anteriores, é reservado um novo bloco que irá apontar para blocos que contêm endereços dos blocos dos arquivos. O número desse novo bloco é armazenado num bloco de endereçamento duplamente indireto
- se isso ainda não for suficiente, é alocado um bloco de endereçamento triplamente indireto.

Acompanhe o processo descrito acima pela figura abaixo:



As vantagens desse método são as seguintes:

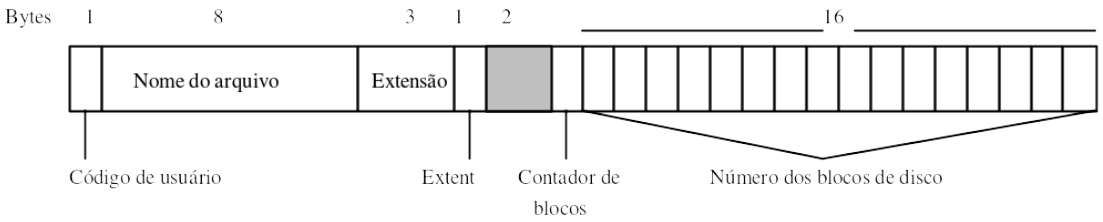
- os blocos indiretos são utilizados apenas quando necessário
- mesmo para os maiores arquivos, no máximo são necessários três acessos a disco para se conseguir o endereço de qualquer de seus blocos

5.2.3. Estrutura do diretório

Devemos estabelecer agora de que forma as informações sobre os arquivos devem ser organizadas no diretório. Para isto veremos três exemplos de diretórios: o do CP/M, do MS-DOS e do UNIX.

CP/M

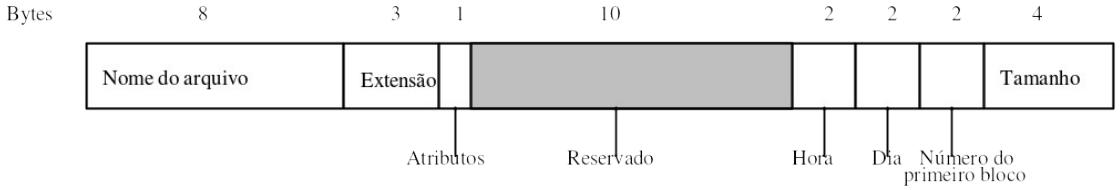
O CP/M possui um único diretório, que consiste de diversas entradas como a mostrada abaixo:



O código de usuário associa cada arquivo a um usuário identificado por um número. O campo *extent* é utilizado para arquivos que ocupam mais de uma entrada no diretório (quando são maiores que o tamanho possível em uma única entrada). Nesse campo temos o número correspondente àquela entrada do arquivo (0 para a primeira entrada, 1 para a segunda, e assim sucessivamente). 16 espaços são reservados para os números dos blocos em cada entrada no diretório. O contador de blocos (*block count*) indica quantos desses 16 espaços reservados estão efetivamente sendo utilizados (quando o arquivo ocupa mais de 16 blocos, uma nova entrada deve ser alocada para o mesmo no diretório, com o valor de *extent* ajustado correspondentemente).

MS-DOS

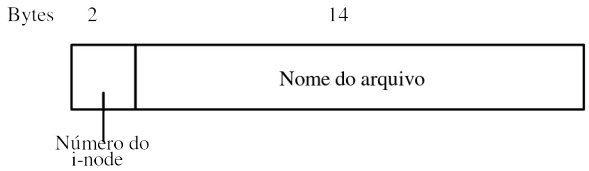
O MS-DOS apresenta uma estrutura como a apresentada na figura abaixo:



O campo de atributos é utilizado para indicar qual o tipo de arquivo (arquivo normal, diretório, escondido, sistema, etc). Um dos campos indica o número do primeiro bloco alocado para o arquivo. Para encontrar o bloco seguinte, busca-se a entrada na FAT correspondente a esse primeiro bloco e encontra-se lá o número do bloco seguinte, repetindo-se esse procedimento até encontrar o fim do arquivo, indicado por um número especial na entrada da FAT. Note que a entrada no diretório de um arquivo apresenta também o tamanho do arquivo. Quando o atributo de um arquivo indica que ele é do tipo diretório, então o mesmo consistirá de entradas como as apresentadas aqui, podendo inclusive apresentar entradas do tipo diretório, possibilitando a formação de uma árvore de subdiretórios.

UNIX

O diretório do UNIX é extremamente simples, devido ao fato de que todas as informações sobre um arquivo estão reunidas no *i-node* desse arquivo, ao invés de no diretório. Cada entrada no diretório é como na figura abaixo:



Os diretórios são arquivos como qualquer outro, com tantos *i-node*s quantos necessários. A raiz tem seu *i-node* armazenado em uma posição fixa do disco. Além dos arquivos normais, cada diretório possui também as entradas "." e "..", geradas no momento da criação do diretório, sendo que a primeira apresenta o *i-node* do próprio diretório e a segunda o *i-node* do diretório pai desse diretório. Para a raiz, "." e ".." são iguais.

5.2.4. Arquivos compartilhados

Quando diferentes usuários se referem a arquivos compartilhados (p.ex., num projeto em grupo), é conveniente que esses arquivos apareçam em diretórios diferentes (os dos diversos usuários) mas mantendo uma única cópia do mesmo em disco (de forma a garantir que quando um usuário faz alterações a esse arquivo, outro usuário terá automaticamente acesso a essas alterações, sem necessitar se preocupar em copiar a nova versão do arquivo). Este processo de associação de um mesmo arquivo a diversos diretórios é chamado de ***link*** (ligação).

Em sistemas cujo diretório possui o endereço dos blocos do arquivo (como é o caso do CP/M), poderíamos realizar a ligação pela cópia dos números de blocos dos diretórios de um usuário para o de outro. O problema aqui é que quando um dos usuários faz uma alteração no arquivo que implique em reorganização dos blocos ocupados por ele, essa alteração não aparecerá ao outro usuário.

Existem duas possíveis soluções para esse problema:

- não listar os números dos blocos no diretório, mas apenas um ponteiro para uma estrutura que lista esses blocos. Este é o caso dos *i-node* do UNIX. Veja que aqui, para estabelecer a ligação, basta fazer no novo diretório onde deve aparecer um arquivo, uma entrada com o nome do mesmo e o número do seu *i-node*.
- podemos criar, no novo diretório, uma entrada de um tipo especial, digamos tipo *link*, que contém, ao invés das informações usuais sobre o arquivo, apenas um *path* para este. Este método é denominado ligação simbólica.

A primeira das soluções apresenta um problema, que pode ser ilustrado da seguinte forma: suponha que o usuário C seja o dono de um arquivo, e então o usuário B faz um *link* para o mesmo. Se permitimos que C apague o arquivo, o diretório de B terá uma entrada que não fará mais sentido. Isto pode ser solucionado (como foi no UNIX), mantendo no *i-node* do arquivo um campo que indica quantos *links* foram realizados para aquele arquivo. Neste caso, quando C remover o arquivo, o S.O. pode perceber que outros usuários ainda querem utilizá-lo e não apaga o *i-node* do arquivo, removendo apenas a entrada correspondente no diretório de C. Isto evita que B fique com um arquivo sem sentido, mas introduz um novo problema: se existirem quotas na utilização de discos (i.e., limites máximos de utilização de disco por cada usuário), teremos o fato de que C continuará pagando pela existência de um arquivo que já não lhe interessa.

A segunda solução (ligação simbólica) não tem esses problemas, pois, quando o dono remove o arquivo o mesmo é eliminado, e a partir desse ponto todos os outros usuários que tentarem acessar o arquivo por ligação simbólica terão a mensagem de "arquivo inexistente". No entanto ela apresenta sobrecarga no tempo de acesso, devido à necessidade de percorrer *paths* extras até se chegar no arquivo, e de

ocupar um *i-node* a mais. Uma outra vantagem desta solução é a de que permite realizar *links* com outras máquinas, interligadas por uma rede (ao contrário do método do *i-node*, que só permite ligações dentro do próprio disco).

As duas soluções apresentadas têm em comum uma desvantagem: o fato de que o arquivo aparece com diferentes *paths* no sistema, fazendo com que, para sistemas automáticos de cópia (em geral em *backups*), ele seja copiado diversas vezes.

Exercícios

5.1. Um sistema operacional suporta apenas um diretório, mas permite que esse diretório tenha um número arbitrário de arquivos, cada qual com nome de tamanho arbitrário. Pode ser simulado um sistema de arquivos hierárquico? Como?

5.2. Dê 5 *path names* diferentes para o arquivo `/etc/passwd`. (Pense sobre as entradas "." e "..").

5.3. Um sistema UNIX usa blocos de 1024 bytes e endereços de disco de 16 bits. O *i-node* armazena 8 endereços de disco para blocos de dados, um endereço de bloco indireto e um endereço de bloco duplamente indireto. Qual é o tamanho máximo de um arquivo nesse sistema?

5.4. Foi sugerido que a primeira parte de um arquivo UNIX fosse armazenada no mesmo bloco de disco do *i-node*. Qual a vantagem disto?

5.5. Um programa UNIX cria um arquivo e escreve no *byte* de número 20 milhões desse arquivo. Qual o espaço de disco ocupado por esse arquivo (incluindo *i-node* e blocos indiretos)? Utilize os valores numéricos do Linux, que são: cada número de bloco ocupa 4 bytes, 7 números de bloco aparecem diretamente no *i-node*, e os blocos de disco são de 1024 bytes.