CAPÍTULO 6 – NÍVEL DE SISTEMA OPERACIONAL

6.1 Introdução

- Nível que automatiza as funções do operador do sistema.

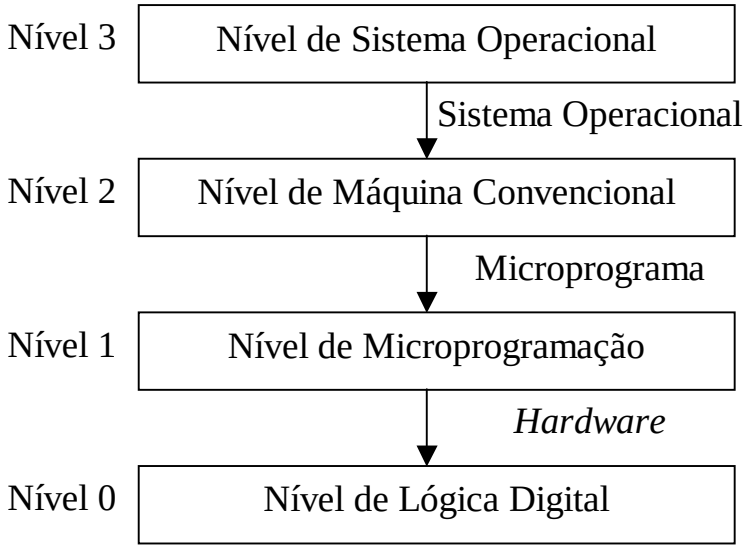

Figura 6.1. O nível de Sistema Operacional.

- Facilidades suportadas pelo nível de sistema operacional:
  - Memória virtual (uso de memória secundária como se fosse memória principal, aumentando o espaço de endereçamento).
  - E/S virtual (instruções de E/S de alto nível facilitam a programação e fornecem garantias mínimas de segurança).
  - Multiprogramação (possibilidade de interpretar várias máquinas de nível 3 em paralelo).
  - Etcétera.

- O Nível de Sistema Operacional inclui as seguintes instruções:
    - Instruções de linguagem de máquina, interpretadas diretamente pelo nível inferior (microprograma ou *hardware*).
    - Instruções adicionais, próprias do nível, interpretadas pelo Sistema Operacional (SO): programa de nível 2 que interpreta o nível 3.

- A interpretação de instruções implica na realização de ciclos de busca-decodificação-execução:
    - PC de Nível 3 aponta para a próxima instrução de nível de SO.
    - PC de Nível 2 aponta para a próxima instrução do SO.
    - PC de Nível 1 aponta para a próxima microinstrução a ser executada pelo *hardware*.

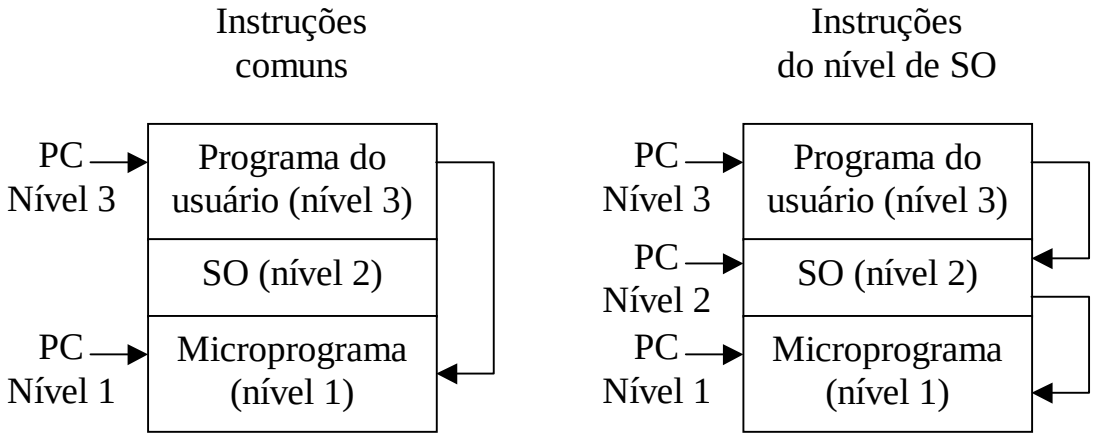

Figura 6.2. Interpretação de instruções comuns e instruções do nível de SO.

6.2 Memória Virtual

- Princípio: uso de memória secundária para simular memória principal.

- Histórico:
  - programas grandes que não cabiam na memória principal eram divididos em blocos (*overlays*) e armazenados em disco.
  - O programador era responsável por criar os blocos e gerenciar a transferência dos mesmos para a memória principal em tempo de execução.
  - Chama-se de Memória Virtual à automatização deste processo em nível de Sistema Operacional.

Paginação:

- Utilização de um Espaço de Endereçamento Virtual maior do que o Espaço de Endereçamento Físico.

  - Espaço de endereçamento físico: conjunto de endereços correspondentes ao espaço de armazenamento disponível na memória principal da máquina.
  - Espaço de endereçamento virtual: conjunto de 2^n endereços que podem ser gerados por uma palavra de endereço de n bits.

- Implementação de um mecanismo de mapeamento entre endereços virtuais e endereços físicos.

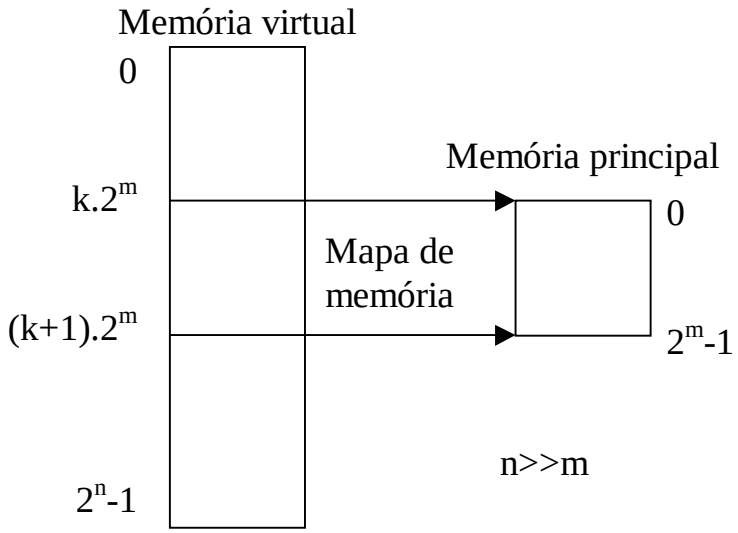

Figura 6.3. Mapeamento entre endereços virtuais e físicos.

- A implementação da paginação é transparente ao programador:
  - As instruções operam sobre endereços virtuais.
  - Para o programador é como se existisse uma memória principal grande, contínua e linear, com o mesmo tamanho do espaço de endereçamento virtual.
  - O programador não precisa se preocupar com o funcionamento da memória virtual.

- Implementação da paginação:

  - O espaço de endereçamento virtual é dividido em páginas virtuais, cujo tamanho é uma potência de 2.
  - O espaço de endereçamento físico ocupa um número inteiro de molduras de páginas.
  - Páginas virtuais são criadas em memória secundária.
  - Programas são armazenados em páginas virtuais.
  - Quando necessário, páginas virtuais são carregadas em molduras de páginas.
  - As páginas virtuais originais devem ser mantidas atualizadas, de acordo com as molduras de página correspondente.

  Exemplo:

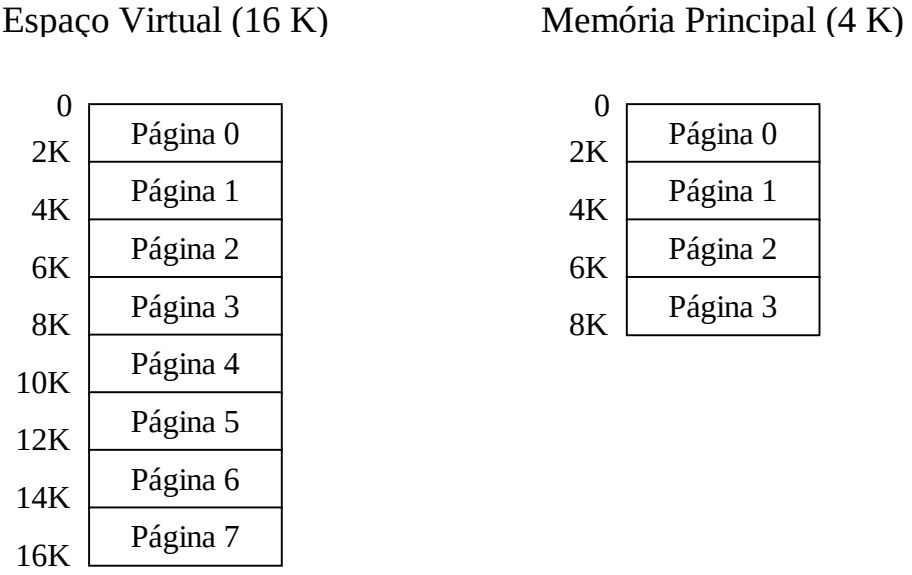

Figura 6.4. - Divisão de um espaço de endereçamento virtual de 16 K em 8 páginas virtuais de 2 K e de um espaço de endereçamento físico de 8 K em 4 molduras de páginas de 2 k.

- Para um espaço de endereçamento virtual de 16 K, temos endereços virtuais de 14 bits. Então, para 8 páginas virtuais de 2 K, o endereço virtual tem 14 bits: 3 bits (Nº de página virtual) + 11 bits (deslocamento dentro da página virtual).
- Para um espaço de endereçamento físico de 8 K, temos endereços físicos de 13 bits. Então, para 4 molduras de páginas de 2 K, o endereço físico tem 13 bits: 2 bits (Nº de moldura de página) + 11 bits (deslocamento dentro da moldura de página).

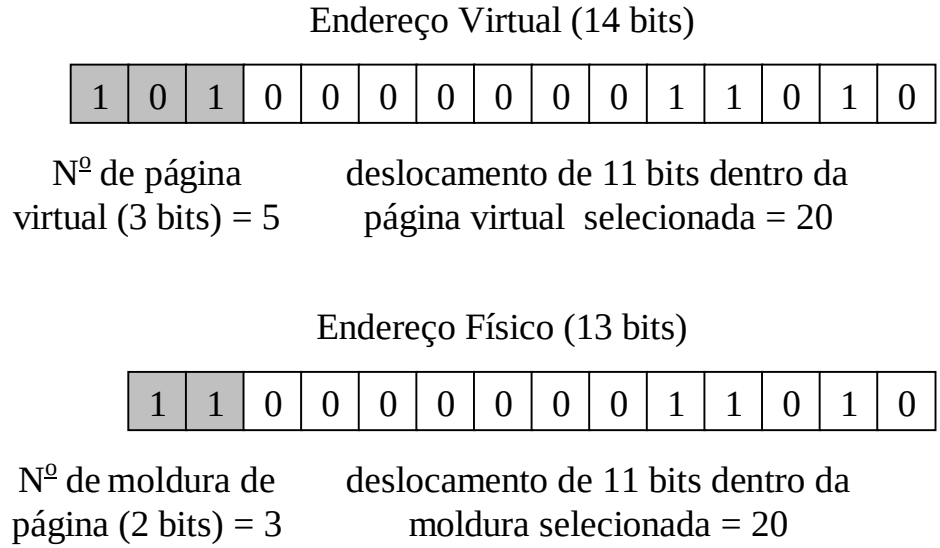

Figura 6.5. - Endereço virtual e endereço físico para memória virtual de 8 páginas de 2 K e memória física de 4 páginas de 2 K.

Mapeamento entre endereços virtuais e endereços físicos:

- Unidade de Gerência de Memória (MMU – *Memory Management Unit*) é o *hardware* responsável por mapear endereços virtuais em endereços físicos.
- A MMU extrai o número de página virtual do endereço virtual e indexa uma Tabela de Páginas, através da qual pode determinar em que moldura de página está armazenada a página virtual em questão.
- A tabela de páginas possui dois campos:
  - Bit de presença/ausência: ativado quando uma página virtual possui cópia em uma moldura de página.
  - Número de moldura de página: contém o número de moldura de página na qual foi copiada a página virtual.

- O endereço físico é obtido pela MMU concatenando o número de moldura de página (obtido da tabela) e o deslocamento dentro da página virtual (obtido do endereço virtual). Este último é igual ao deslocamento dentro da moldura de página.

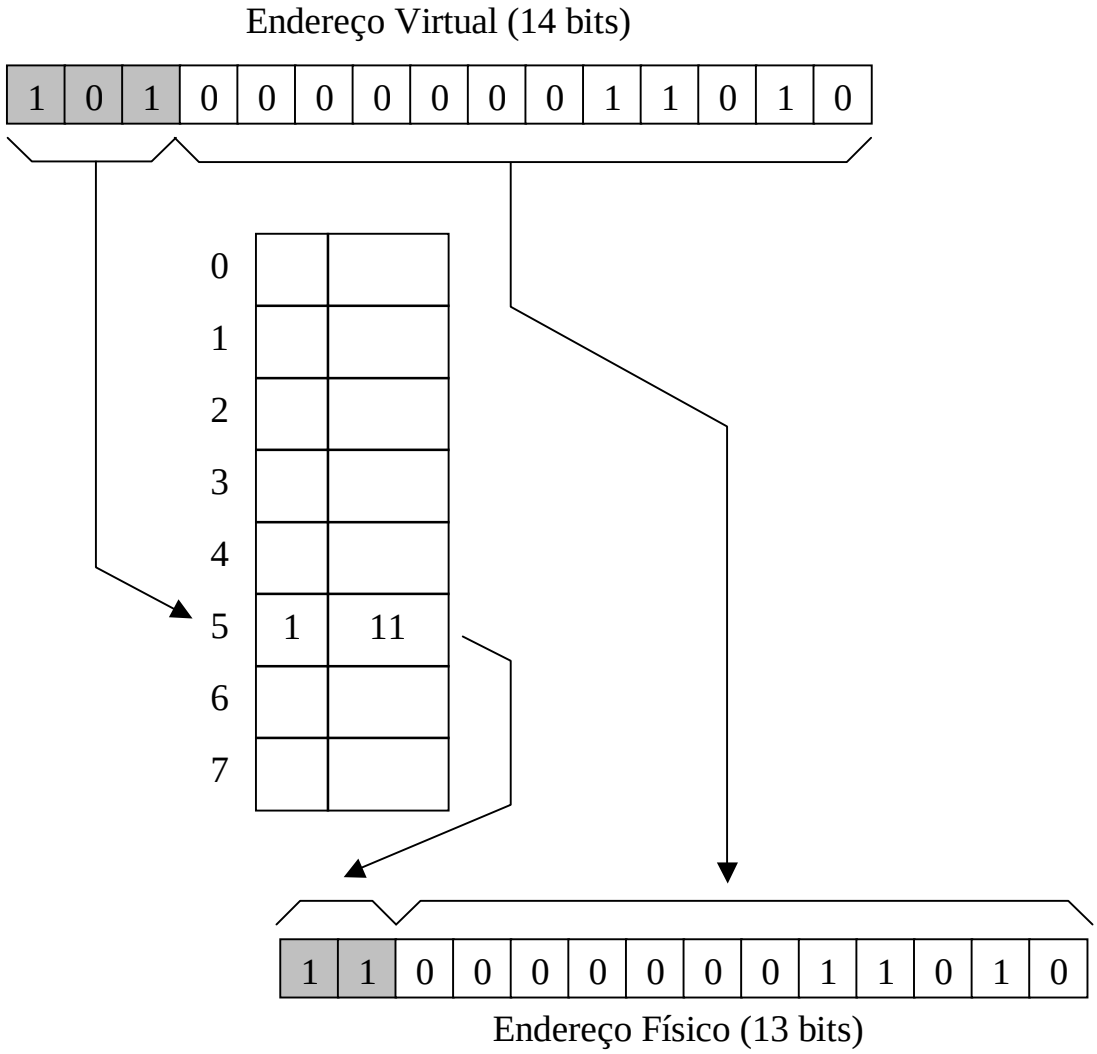

Figura 6.6. – Mapeamento entre endereços virtuais e endereços físicos através da tabela de páginas.

Paginação por demanda:

- Quando uma referência a um endereço virtual não encontra a palavra procurada numa moldura de página na memória principal (bit de presença/ausência desativado), diz-se que ocorre uma Falta de Página.
- Na Paginação por Demanda, as páginas virtuais são carregadas em molduras de páginas quando são referenciadas explicitamente.
- Ao acontecer uma falta de página, o sistema operacional copia a página virtual correspondente para uma moldura de página, atualiza a tabela de páginas com esta informação e repete a instrução de referência à memória que produziu a falta.

- Problema: em regime de compartilhamento de tempo, se existir muito chaveamento entre programas, reiniciar um processo interrompido implica em recarregar páginas já carregadas e posteriormente expurgadas da memória principal, deteriorando o desempenho.

Paginação por conjunto de trabalho:

- Programas não referenciam uniformemente seu espaço de endereçamento. As referências tendem a se agrupar em um pequeno número de páginas. No início da execução
- Conjunto de Trabalho (CT): conjunto de páginas usadas pelas k referências mais recentes à memória num dado instante t. Após carregar um certo número de páginas, o conjunto de trabalho muda lentamente.

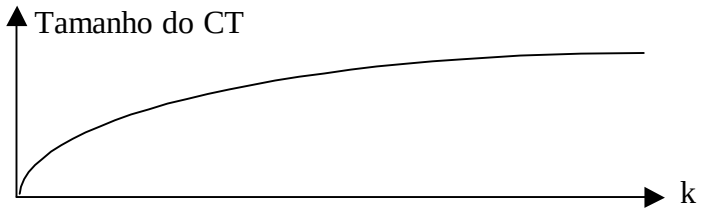

Figura 6.7. – Conjunto de trabalho em função das k referências mais recentes à memória.

- As páginas que provavelmente serão necessárias quando um programa for recomeçado são trazidas previamente para a memória principal, enquanto algum outro programa estiver sendo executado.
- No transitório, quando o conjunto de trabalho ainda não está estabilizado, páginas desnecessárias podem vir a ser carregadas.

Política de Substituição de Páginas:

- Quando a memória principal está cheia e é feita uma referência a uma página virtual que ainda não tem cópia em uma moldura de página, é necessário carregá-la em alguma moldura ocupada, expurgando o seu conteúdo atual.

⇒ Deve-se fazer uma previsão sobre qual página será menos necessária.

Algoritmo LRU (*Least Recently Used*): consiste em substituir a página que foi utilizada menos recentemente.

- Se a memória principal exceder o tamanho do conjunto de trabalho, o algoritmo LRU tende a minimizar o número de faltas de página.
- Problema: em situações específicas pode falhar desastrosamente. Exemplo: dado uma memória principal dividida em n molduras de páginas, tentar o acesso repetido (em laço) a uma seqüência de (n+1) páginas virtuais.
- A ocorrência freqüente de faltas de páginas é chamada de *Trashing*. Esta situação ocorre quando o conjunto de trabalho é maior do que o número de molduras disponíveis.

Algoritmo FIFO (*First-In, First-Out*): consiste em substituir a página que foi carregada menos recentemente, ou seja, a primeira a entrar é a primeira a sair. A substituição independe da última referência feita à página.

- Cada moldura de páginas tem um contador associado, inicialmente zerado.
- A cada falta de página, os contadores são incrementados.
- O contador associado a uma moldura no qual foi carregada uma nova página é zerado.
- Quando a memória enche (todos os bits de presença/ausênia iguais a 1), ao ocorrer uma falta, a página com maior contador é expurgada da sua moldura atual.

- Atualização de Páginas Virtuais:
  - Quando uma página na memória principal é modificada através de uma escrita, é necessário alterar a página virtual original no disco.
  - MMU pode incluir um bit (*Dirty Bit*) por página para distinguir as páginas alteradas (páginas sujas) daquelas que não foram alteradas (limpas). Através do mesmo, o sistema operacional pode determinar a necessidade de atualização ou não da página virtual original.

Fragmentação:

- Programas geralmente ocupam um número não inteiro de páginas. ⇒ Em média, desperdiça-se a metade da última página de cada programa.
- Este desperdiço de memória é chamado de Fragmentação Interna, pois o espaço perdido localiza-se dentro da última página.
- Conclusão: o tamanho da página deve ser projetado cuidadosamente.
  - Páginas grandes:
    - Maior fragmentação (uso menos eficiente do espaço de armazenamento).
  - Páginas pequenas:
    - Menor fragmentação (uso mais eficiente do espaço de armazenamento).
    - Menos *Trashing* quando o conjunto de trabalho constituído por muitas páginas dispersas no espaço de endereçamento virtual.
    - Maior número de páginas. ⇒ Maior tabela de páginas. ⇒ Mais registradores na pastilha. ⇒ Processador mais caro.
    - Uso menos eficiente da banda passante do disco. (Transferências de blocos maiores são mais eficientes do que transferências de blocos menores, devido a que ambas ocupam o mesmo tempo médio de busca mais tempo de latência rotacional).

Segmentação:

- Problemas com a Paginação: o uso de um espaço de endereçamento unidimensional único pode resultar em que estruturas de dados dinâmicas (pilhas, tabelas de símbolos, etc.) encostem umas nas outras.
- Solução: uso de Segmentação (divisão da memória em segmentos).
- Segmentos: Espaços de endereçamento unidimensionais independentes de tamanho variável.
- Cada segmento é constituído por uma seqüência linear de endereços começando sempre em zero e variando até um endereço máximo.
- A organização da memória é bidimensional. A especificação de um endereço deve incluir o número do segmento e o endereço da palavra dentro do espaço de endereçamento do segmento.
- O segmento armazena um único tipo de estrutura de dados. Ao contrário da paginação, não é transparente ao usuário.

- Vantagens da segmentação:

  - A segmentação permite manipular de forma independente estruturas cujos comprimentos variam.
  - Segmentos diferentes podem ter tipos diferentes de proteção. Exemplo: segmento que armazena código executável deve ser apenas para leitura.
  - O processo de ligação é simplificado. Exemplo: segmentos podem armazenar procedimentos que sempre começam no endereço zero; a chamada a um dado procedimento faz referência ao número de segmento no qual está armazenado.
  - Se um procedimento for modificado, nenhum outro precisa ser modificado, ao contrário do que acontece em uma memória paginada.
  - A segmentação facilita o compartilhamento de procedimentos entre vários processos paralelos, evitando a necessidade de manter cópias do mesmo procedimento para cada processo faz uso do mesmo.

Aspecto	**Paginação**	**Segmentação**
Para que foi inventada esta técnica?	Para simular uma memória grande.	Para permitir o uso de múltiplos espaços de endereçamento.
Quantos espaços de endereçamentos lineares existem?	Um.	Vários.
O espaço de endereçamento virtual pode exceder o tamanho da memória principal?	Sim.	Sim.
Tabelas de tamanho variável podem ser tratadas com facilidade?	Não.	Sim.
O programador precisa saber que o sistema implementa esta técnica?	Não.	Sim.

Figura 6.8. – Comparação entre paginação e segmentação.

Implementação da Segmentação:

- Segmentação por *Swapping*:
  - Semelhante à paginação por demanda. Diferença: páginas têm tamanho fixo, enquanto que o tamanho dos segmentos é variável.
  - A memória principal armazena um conjunto de segmentos.
  - Quando ocorrer uma referência a um segmento que não está na memória principal, deverá ser carregado na mesma a partir do disco.
  - Se não houver espaço na memória principal, um ou mais segmentos devem ser expurgados para abrir espaço para o novo.
  - Se um segmento expurgado não possuir um original "limpo" em disco, este deve ser atualizado.
  - O espaço aberto ao expurgar um ou mais segmentos deve ser suficientemente grande, de modo a conter o novo segmento.
  - O espaço aberto geralmente é maior do que o novo segmento, gerando um "buraco" não utilizado na memória.
  - Após um certo tempo, a memória contém um grande número de regiões contendo segmentos separadas por regiões não utilizadas. Este fenômeno é denominado "Tabuleiro de Xadrez" (*checkerboarding*). Isto produz Fragmentação Externa (o espaço perdido localiza-se fora dos segmentos).

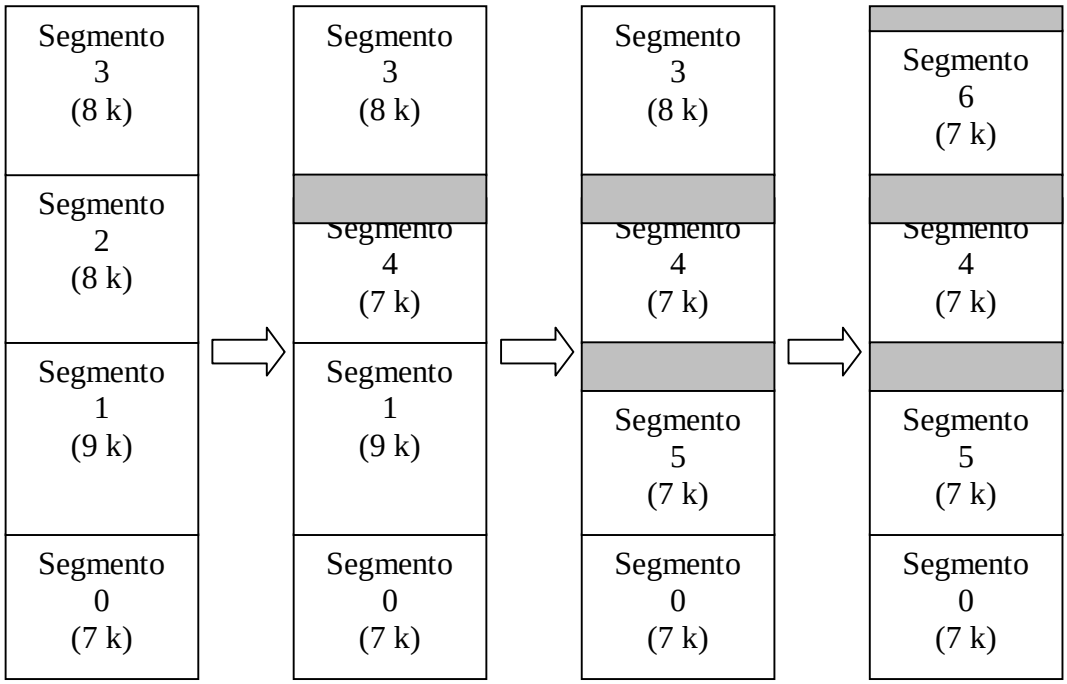

Figura 6.9. – Fragmentação externa: formação do tabuleiro de xadrêz.

- Técnicas para contornar a fragmentação externa:
  - Compactação: mover todos os segmentos que estiverem acima de um buraco de modo a ocupar o espaço do mesmo.
    - a) Compactar sempre que aparece um buraco.
    - b) Quando existir uma grande percentagem de buracos.
  - Visa abrir um grande buraco no final do espaço de endereçamento.
  - Desvantagem: gasta muito tempo para ser feita.
  - Melhor Assentamento (*Best Fit*): Colocar um novo segmento no buraco no qual caiba com a menor folga possível.
    - Evita que buracos grandes (que podem abrigar segmentos grandes) sejam desmembrados em buracos menores.
    - É necessário manter uma lista de buracos, contendo os seus endereços e seus tamanhos correspondentes.
  - Primeiro Assentamento (*First Fit*): Colocar um novo segmento no primeiro buraco com espaço suficiente encontrado.
    - É necessário manter uma lista de buracos, contendo os seus endereços e seus tamanhos correspondentes.
    - Mais rápido que o *Best Fit*.
    - Melhor desempenho do que o *Best Fit*, pois este último tende a gerar um grande número de buracos pequenos inúteis.
  - Compensação de Buracos: ao retirar um segmento vizinho a um buraco, eliminar este último criando um buraco maior, constituído pela soma do buraco velho mais o novo buraco.
    - É necessário manter uma lista de buracos, contendo os seus endereços e seus tamanhos correspondentes.
    - Pode ser utilizado associado a *Best Fist* ou *Fisrt Fit*, sempre que uma certa percentagem de buracos for criada.

- Segmentação por Paginação:
  - Segmentos são divididos em um conjunto páginas.
  - Páginas são carregadas por demanda na memória principal.
  - Em um dado momento, algumas das páginas de um segmento podem estar na memória principal e outras no disco.
  - É necessário manter uma tabela de páginas para cada segmento.

Comparação entre Memória Virtual e Memória Chache:

- A memória cache e a memória virtual apresentam muitas semelhanças e utilizam técnicas análogas, diferindo no nível hierárquico ao qual se aplicam.

Aspecto de comparação (Semelhanças)	Memória Cache	Memória Virtual
Armazenamento do programa original	Programa mantido em memória principal.	Programa mantido em disco.
Divisão do programa	Blocos de cache de tamanho fixo.	Páginas de tamanho fixo.
De que forma e onde são armazenadas as informações acessadas mais freqüentemente?	Subconjunto de blocos na memória cache.	Subconjunto de páginas em molduras na memória principal.
Como melhoram o desempenho?	Se um programa usar os blocos no cache com muita freqüência, serão geradas poucas falhas e o programa rodará rápido, acessando pouco a memória principal.	Se um programa usar páginas na memória principal com muita freqüência, serão geradas poucas faltas de página e programa rodará rápido, acessando pouco o disco.

Aspecto de comparação (Diferenças)	Memória Cache	Memória Virtual
Nível hierárquico em que atuam	*Hardware* e microprogramação.	Sistema Operacional.
Tamanho das partes em que é dividido o programa	Menor. Exemplo: blocos de 64 *bytes*.	Maior. Exemplo: páginas de 8 KB.
Mapeamento de endereços	Índice de Cache constituído pelos bits menos significativos do endereço.	Tabela de páginas indexada pelos bits mais significativos do endereço.

Figura 6.10. – Comparação entre memória Cache e Memória Virtual.

6.3 Multiprogramação

Suporte de várias máquinas virtuais de nível 3 rodando em paralelo

Justificativa:

- Processamento paralelo: alguns algoritmos podem ser mais bem programados por meio de dois ou mais processos cooperantes rodando em paralelo.
- Sistemas de tempo compartilhado: várias pessoas podem compartilhar simultaneamente os recursos de um dado computador.

Implementação:

- Paralelismo real: o computador possui mais de um processador. O sistema operacional deve gerenciar a distribuição das tarefas entre os diversos processadores.
- Paralelismo simulado: o computador possui um único processador. O sistema operacional deve multiplexar no tempo a execução dos processos paralelos.

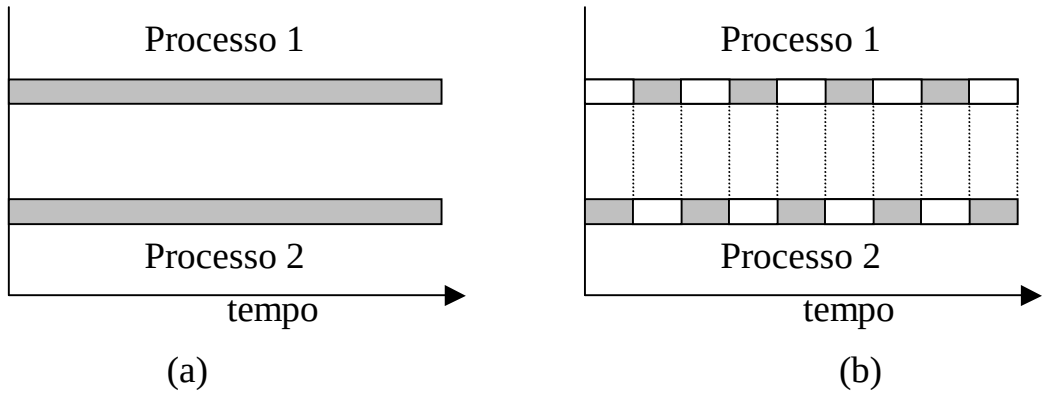

Figura 6.11. – a) Paralelismo real. b) Paralelismo simulado.

- Programa roda como parte de um Processo = programa + espaço de endereçamento + estado (valor de PC, SP, PSW, etc.).
- Sistemas operacionais fornecem instruções de nível 3 (chamadas) para criar parar, examinar reiniciar e terminar a execução de processos dinamicamente.
- Por meio destas instruções, processos pais podem criar processos filhos, podendo ter controle total, parcial ou nenhum sobre os mesmos.

Exemplo de processos paralelos – Corrida Crítica:

- Dois processos, Produtor e Consumidor, executados assincronamente, a velocidades diferentes, comunicando-se por meio de um *buffer* circular compartilhado na memória principal.
- Produtor: calcula números e os armazena no *bufer*, um de cada vez.
- Consumidor: remove os números do buffer, um de cada vez e os imprime.
- *Buffer* circular:
  - Ponteiro *entrada*: aponta para a próxima palavra livre no *buffer* (onde o produtor colocará o próximo número primo produzido).
  - Ponteiro *saída*: aponta para o próximo número a ser removido pelo consumidor.
  - O topo do *buffer* é logicamente contíguo ao fundo do mesmo. (Se um ponteiro aponta para a posição do fundo do *buffer*, o seu incremento deve retornar o endereço da posição no topo).
  - Quando *entrada* = *saída*, o *buffer* está vazio.
  - O *buffer* é considerado cheio quando todas as suas palavras, exceto uma, armazenarem um número primo. Nesta situação, *entrada* está uma palavra atrás (logicamente) de *saída*.

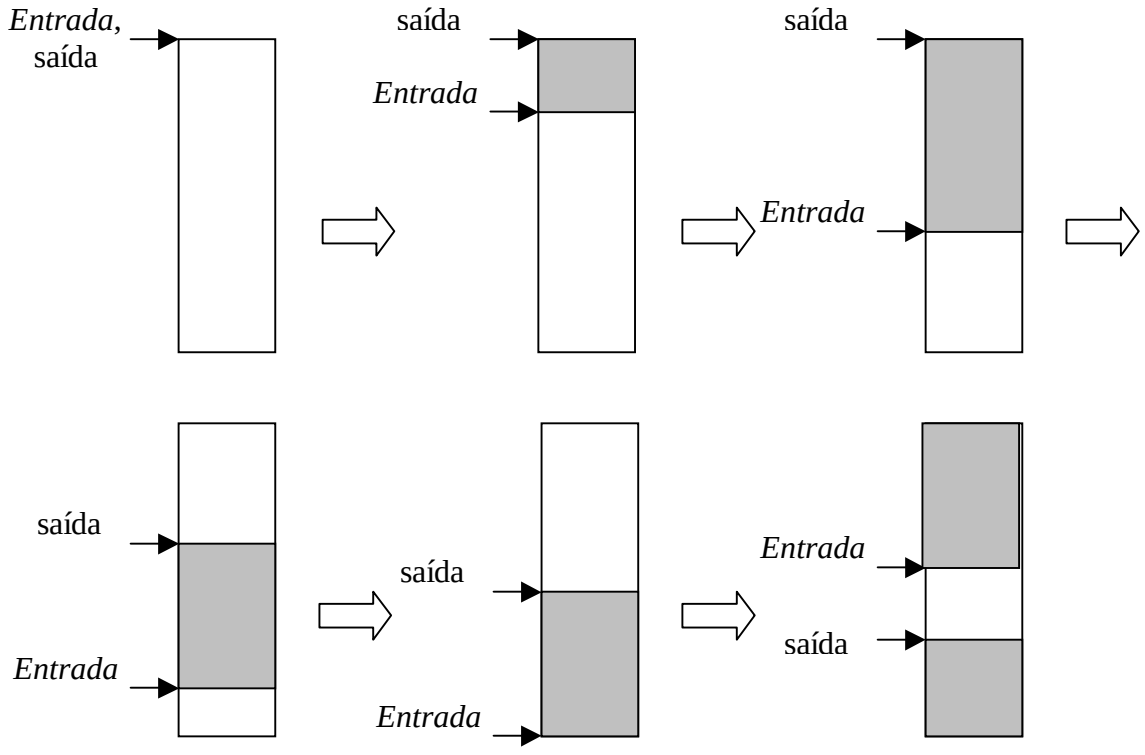

Figura 6.12. – Evolução típica de um *buffer* circular.

- Procedimentos utilizados pelos processos Produtor e Consumidor:
  - PróximoPrimo(x): retorna o número primo seguinte ao número primo x.
  - Imprima(x): imprime o número x.
  - Próximo(ponteiro): incrementa logicamente ponteiro.
  - Durma: coloca o processo para dormir.
  - Acorde(P): acorda um processo P que estiver dormindo.

Procedimento Produtor
{Calcula números primos e os coloca no *buffer* compartilhado. Quando o *buffer* está cheio, o Produtor vai dormir, retomando a sua operação quando o Consumidor enviar um sinal de acordar}.

```
Começar
   primo := 1;
   Enquanto primo < maxprimo faça
   Começar
      {P1} primo := PróximoPrimo(primo);
      {P2} se Próximo(entrada) = (saída) então Durma;
      {P3} buffer[entrada] := primo;
      {P4} entrada := Próximo(entrada);
      {P5} se Próximo(saída) = entrada então Acorde(Consumidor);
   fim
fim;
```

Procedimento Consumidor
{Retira números do *buffer* e os imprime. Se o *buffer* estiver vazio, o consumidor vai dormir, retomando a sua operação quando o Produtor enviar um sinal de acordar}.

```
Começar
   omirp := 1;
   Enquanto omirp < maxprimo faça
   Começar
      {C1} se entrada = saída então Durma;
      {C2} omirp := buffer[saída];
      {C3} saída := Próximo(saída);
      {C4} se saída = Próximo(Próximo(entrada)) então Acorde(Produtor);
      {C5} Imprima(omirp);
   fim
fim;
```

Figura 6.13. – Processos paralelos com corrida crítica.

Problema: Corrida Crítica.

- Considere que em um dado momento:
  - Somente um número foi deixado no *buffer* na posição 32.
  - *entrada* = 33, *saída* = 32.
  - O Produtor está no comando P1, calculando um novo primo.
  - O Consumidor está em C5, imprimindo o número da posição 31.
- A seguir, o Consumidor:
  - Termina a impressão do número.
  - Faz o teste em C1 (*entrada* = 33 ≠ *saída* = 32) ⇒ *buffer* não vazio. Ainda não deve dormir.
  - Tira o último número do *buffer* em C2.
  - Incrementa *saída* (*saída* = 32+1 = 33) em C3.
  - Faz o teste em C4 (*saída* = 33 ≠ *entrada*+2 = 35) ⇒ o *buffer* não estava vazio. Não é necessário acordar o Produtor.
  - Imprime o último número em C5.
  - Inicia o passo C1, indo procurar na memória os ponteiros *entrada* e *saída*, necessários para o teste de *buffer* vazio. Repare que estes valores são iguais neste momento.
- Suponha que após o Consumidor buscar os ponteiros na memória, mas antes de compara-los, o Produto faz o seguinte :
  - Conclui o cálculo do novo primo, finalizando P1.
  - Faz o teste em P2 (*entrada*+1 = 34 ≠ *saída* = 33) ⇒ *buffer* não cheio. Ainda não deve dormir.
  - Em P3, armazena o novo número no *buffer*, na posição 33.
  - Em P4, incrementa *entrada* (*entrada* = 33+1 = 34).
  - Faz teste em P5 (*saída*+1 = 34 = *entrada*) ⇒ Verdade, o *buffer* estava vazio, de modo que envia um sinal de acordar para o Consumidor (assumindo que o mesmo deveria estar dormindo, o que é falso). O sinal de acordar é perdido.
  - O consumidor começa a procurar um novo primo em P1.
- O consumidor retoma a execução:
  - Continuando o passo C1, realiza o teste usando os valores dos ponteiros que já tinha buscado na memória. Como os valores buscados são iguais, o Consumidor vai dormir.
- O consumidor retoma a execução:
  - O processo consumidor prossegue a sua operação até encher o *buffer*, quando se coloca para dormir, esperando que o consumidor o acorde.
- **Resultado**: os dois processos ficam dormindo, esperando por um sinal de acordar que não vai chegar nunca.

Solução para o problema de Corrida Crítica:

a) Equipar cada processo com um bit de "espera acordado":
   - Este bit é ativado sempre que um processo ainda em execução recebe um sinal de acordar.
   - Quando o processo vai dormir e este bit estiver ligado, volta imediatamente a ser executado, zerando o bit.
   - Falha quando existem n processos diferentes (n>2). Uma solução desajeitada seria manter (n-1) bits de espera "acordado" por processo.

b) Usar Semáforos:

   - Esta técnica permite sincronizar um número arbitrário de processos.
   - Um semáforo permite armazenar sinais de acordar para uso futuro, de modo a não perde-los.
   - Um semáforo é uma variável inteira não negativa.
   - Duas instruções de nível de sistema operacional operam sobre semáforos:
     - DOWN: Se o semáforo é maior do que zero, decrementa o semáforo. Se o semáforo é zero, o processo que estiver realizando o DOWN é colocado para dormir e permanece assim até que outro processo realize um UP no semáforo.
     - UP: Incrementa o semáforo. Se o semáforo estava em zero e outro processo estava dormindo por causa dele, este processo pode completar a operação DOWN que o suspendeu, zerando o semáforo e retomando a sua execução.
   - Propriedade das instruções UP e DOWN: atomicidade – uma vez que um processo tiver iniciado uma instrução em um semáforo, nenhum outro processo poderá acessa-lo até que o primeiro tiver finalizado a sua instrução.

Instrução	**S = 0**	**S > 0**
DOWN	O processo é suspenso até que outro processo realize um UP em S.	S = S - 1
UP	S = S + 1. Se outro processo foi suspenso tentando completar um DOWN em S, o mesmo deve completar o DOWN e retomar a execução.	S = S + 1

Figura 6.14. – Instruções sobre um semáforo S.

Procedimento Produtor
{Calcula números primos e os coloca no *buffer* compartilhado. Quando o *buffer* está cheio, o Produtor vai dormir realizando um DOWN em Disponível. Quando o Consumidor realiza um UP em Disponível, o produtor retoma a execução em P3}.

```
Começar
   primo := 1;
   Enquanto primo < maxprimo faça
   Começar
      {P1} primo := PróximoPrimo(primo);
      {P2} DOWN(Disponível);
      {P3} buffer[entrada] := primo;
      {P4} entrada := Próximo(entrada);
      {P5} UP(Preenchido);
   fim
fim;
```

Procedimento Consumidor
{Retira números do *buffer* e os imprime. Se o *buffer* estiver vazio, o consumidor vai dormir realizando um DOWN em Preenchido. Quando o Produtor realiza um UP em Preenchido, o consumidor retoma a execução em C2}.

```
Começar
   omirp := 1;
   Enquanto omirp < maxprimo faça
   Começar
      {C1} DOWN(Preenchido);
      {C2} omirp := buffer[saída];
      {C3} saída := Próximo(saída);
      {C4} UP(Disponível);
      {C5} Imprima(omirp);
   fim
fim;
```

Figura 6.15. – Processos paralelos sem corrida crítica, usando semáforos.

Comportamento na situação de Corrida Crítica:

- Considere que:
  - Tamanho do *buffer* = 100.
  - Inicialmente, Preenchido = 0, Disponível = 100.
- Início da operação:
  - Produtor começa em P1.
  - Consumidor começa em C1, dando um DOWN em Preenchido, o que suspende o Consumidor, visto que Preenchido = 0.
  - Após encontrar o primeiro primo, o Produtor, em P2, executa um DOWN em Disponível (Disponível = 100 –1 = 99).
  - A seguir, o produtor armazena o novo primo no *buffer* (P3) e incrementa o ponteiro entrada (P4).
  - Em P5, o Produtor executa um UP em Preenchido (Preenchido = 0+1 = 1), liberando o Consumidor, que pode completar o seu DOWN, assim, Preenchido = 1-1 = 0. Ambos os processos estão em execução.
- Corrida crítica:
  - somente um número foi deixado no *buffer* na posição 32.
  - *entrada* = 33, *saída* = 32. Preenchido = 1, Disponível = 99.
  - O Produtor está no comando P1, calculando um novo primo.
  - O Consumidor está em C5, imprimindo o número da posição 31.
  - Em C1, o Consumidor termina a impressão do número e inicia um DOWN em Preenchido (Preenchido = 1-1 = 0).
  - O Consumidor retira o último número do *Buffer* em C2, incrementa o ponteiro *saída* em C3 e, em C4, faz um UP em Disponível (Disponível = 99+1 = 100). Em C5, imprime o primo e vai para C1.
  - Exatamente antes do Consumidor realizar o DOWN em Preenchido, o Produtor finaliza o cálculo de um novo primo e executa P2 (Disponível = 100-1 = 99), P3 e P4.
  - Preenchido = 0. Neste semáforo, o Produtor está prestes a realizar um UP e o Consumidor está prestes a realizar um DOWN.
    - Se o Consumidor executar o DOWN primeiro, ele será suspenso até que o produtor o libere realizando um UP.
    - Se o Produtor executar o UP primeiro, o semáforo será incrementado (Preenchido = 0+1 = 1) e o Consumidor não será mais suspenso.

    $\Rightarrow$ Nenhum sinal de acordar é perdido.