# **Unidade 09**: Sistemas Operacionais

## Gerenciamento de Processos e Memória

Prof. Daniel Caetano

**Objetivo**: Apresentar a lógica básica do gerenciamento de processos e memória, destacando os aspectos de interesse ao engenheiro eletrônico.

**Bibliografia**:

- MACHADO, F. B (2007); TANENBAUM, A. S. (2003); STALLINGS, W. (2003); MURDOCCA, M. J. (2000).

## INTRODUÇÃO

Na aula anterior vimos, em linhas gerais, a importância do Sistema Operacional no uso de máquinas processadas.

Nesta aula serão apresentados maiores detalhes sobre o gerenciamento de processos e o gerenciamento de memória, destacando os aspectos que devem ser preocupação do engenheiro eletrônico projetista de hardware.

## 1. GERENCIAMENTO DE PROCESSOS

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

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

Entretanto, a partir do momento em que os computadores (e sistemas operacionais) passaram a executar diversos programas ao mesmo tempo, surgiu uma necessidade de criar uma "separação" entre estas tarefas, e dessa necessidade surgiu o conceito de *processo*, que se tornou unânime. Neste contexto, o conceito de processo é o mais importante de um sistema operacional moderno.

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

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

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

## 1.1. O Modelo de Processo (Opcional)

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

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

Estas informações (sobre quanta RAM está disponível em um processo, em que ponto ele está em sua execução, qual seu estado e prioridades etc.) são armazenadas em uma estrutura chamada "Bloco de Controle de Processos" (Process Control Block - PCB). A figura 1 apresenta um exemplo de estrutura de PCB:

| Apontador | Estado do Processo |
|---|---|
| Número do Processo | |
| Contador de Instruções | |
| Registradores | |
| Limites de Memória | |
| Lista de Arquivos Abertos | |
| ... | |

**Figura 1**: Exemplo de estrutura de um PCB

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

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

### 1.1.1. Estado do Hardware (Opcional)

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

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

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

### 1.1.2. Contexto do Software (Opcional)

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

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

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

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

### 1.1.3. Configuração de Memória (Opcional)

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

## 1.2. Estados de Processo

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

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

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

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

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

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

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

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

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

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

### 1.2.1. Mudanças de Estado

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

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

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

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

### 1.3. Escalonamento de Processos (*Process Scheduling*)

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

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

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

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

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

## 2. GERENCIAMENTO DE MEMÓRIA

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

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

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

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

### 2.1. Alocação Contígua Simples

* Sistemas monoprogramados
  - Existe isso?
* Dois processos: Programa + SO
  - Figura 3
* Processo: Respeitar o limite de memória
* Primeiras CPUs: nenhum controle de acesso
* CPUs posteriores: MMU: Memory Management Unit
  - Registrador que indica fim da área de sistema (Figura 4)
    + Programa do usuário não tem acesso lá!
      = *Access Violation* ou GPF
* Problema: ineficiente e inviável para multitarefa

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

Neste tipo de sistema, em geral, o sistema operacional é carregado em uma dada região da RAM (normalmente a parte mais baixa) e o programa sendo executado (o processo) é carregado em seguida ao sistema operacional, sempre na mesma área (Figura 2). O processo so precisa se preocupar em não ultrapassar o limite de memória disponível.

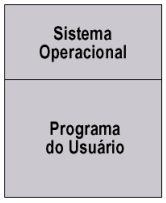


**Figura 2**: Alocação Contígua Simples

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

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

Nestes primeiros sistemas o único controle existente na MMU era um registrador que indicava o endereço de memória onde acabava a área de sistema, ponto em que se iniciáva a área do software do usuário (Figura 3).

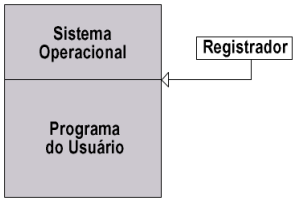


**Figura 3**: Separação entre Área de Sistema e Área de Usuário nas primeiras MMU

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

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

## 2.2. Alocação com Particionamento Dinâmico

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

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

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

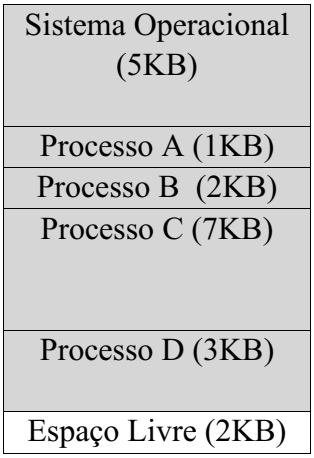


**Figura 4**: Particionamento Dinâmico da RAM

Este modelo, entretanto tem um problema chamado de **fragmentação**, que surge quando processos são inciados e finalizados. Suponha que, ao carregar os programas do

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

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

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

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

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

A primeira solução que foi dada para este problema era "por software", isto é, foi criado um sistema de gerenciamento de particionamento dinâmico com relocação, em que os processos são relocados quando necessário (Figura 6), criando espaço para o novo processo. Entretanto, este processo de relocação é bastante custoso computacionalmente (mesmo nos tempos atuais). Para evitar isso, os sistemas usam estratégias para minimizar a fragmentação da memória, **além de contar com mecanismos em hardware** para tanto.

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

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

## 2.3. Arquivo de Troca (*Swapping*)

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

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

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

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

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

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

se tiver de ser executado por software (além de depender que o programa tenha sido preparado para isso).

## 2.4. Endereçamento Virtual

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

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

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

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

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

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

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

### 2.4.1. Espaço Virtual de Endereçamento

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

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

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

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

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

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

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

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

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

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

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

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

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

| Posição de Memória Referenciada |
|---|
| 0 ~ 127 |
| 128 ~ 255 |

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

Para o processo, ele está em uma região contígua da memória principal, dos endereços 0 ao 255. Entretanto, na prática, ele está parte alocado na região 256 ~ 383 e 640 ~ 767. Este mecanismo se chama *espaço de endereçamento virtual com mapeamento de páginas*. Entretanto, cada processo precisa possuir uma tabela destas. E, de fato, esta tabela faz parte das informações do processo, dados estes armazenados na região de configuração de memória.

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

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

## 2.5. Memória Virtual (Opcional)

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

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

Considere que um sistema tem 1GB de memória principal física (RAM) e vários GBs de disco. A idéia é que um registrador de endereçamento base **possa apontar posições de**

**memória maiores do que a memória física**. Para o processo, ele é acarregado como se o sistema de fato tivesse mais memória disponível do que a física. Toda a parte do processo que ultrapassar a memória física disponível, será automaticamente jogada para o arquivo de troca (swap). Em outras palavras, parte do processo será colocada no disco.

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

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

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

## 3. BIBLIOGRAFIA

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

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

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

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

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.