Gestão de Memória

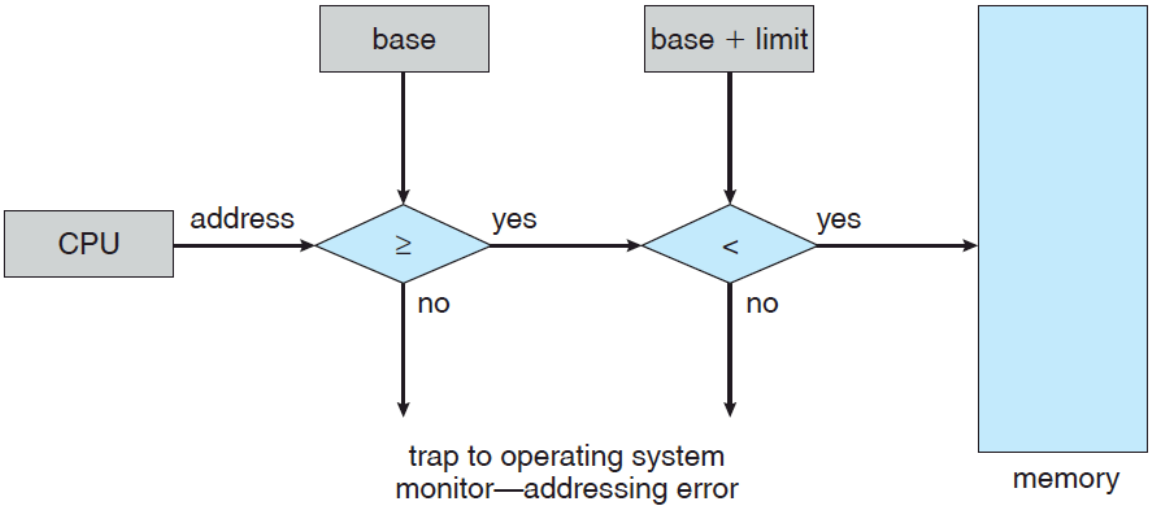
O ponto essencial da gestão de memória é o conceito de memória virtual. Para isso, numa primeira fase irão ser explicadas algumas estratégias de gestão de memória que, posteriormente, serão usadas para implementação da memória virtual. É de notar que o conceito de memorai virtual não quer dizer que temos mais memória do que aquela que a máquina fornece, é apenas uma estratégia que permite, com a memória existente, simular uma memória maior do que aquela que é fornecida. Ou seja, a memória virtual, não é uma memória física, mas sim uma memória conceptual.

É muito importante, por razões de segurança, proteger o acesso ao sistema operativo por parte de processos do utilizador, bem como proteger os processos de utilizadores um dos outros. Esta proteção é fornecida pelo *hardware*.

É necessário garantir que cada utilizador tenha o seu espaço de endereçamento e, para isso, é necessário definir quais os endereços válidos que um processo pode aceder e garantir que um processo apenas tem acesso a esses endereços. Esta garantia pode ser assegurada através de dois registos, um base e um limite. O endereço base é o endereço válido mais baixo do programa, onde o mesmo começa. O limite é a gama de endereços válidos desse programa.

```
Base  = 300040
Limit = 120900
Range = [300400, 420939] = (endereços válidos de um processo)
```

A proteção do espaço de endereçamento é garantida fazendo com que o *CPU* verifique todos os endereços gerados pelos programas em *user mode.* Apenas o sistema operativo consegue aceder aos registos de base e limite pois tem acesso ao *kernel mode* e, com isso, consegue também definir esses mesmos registos. Sempre que um programa em *user mode* tenta aceder a endereços fora do seu espaço de endereçamento, é lançada uma *trap* ao sistema operativo.



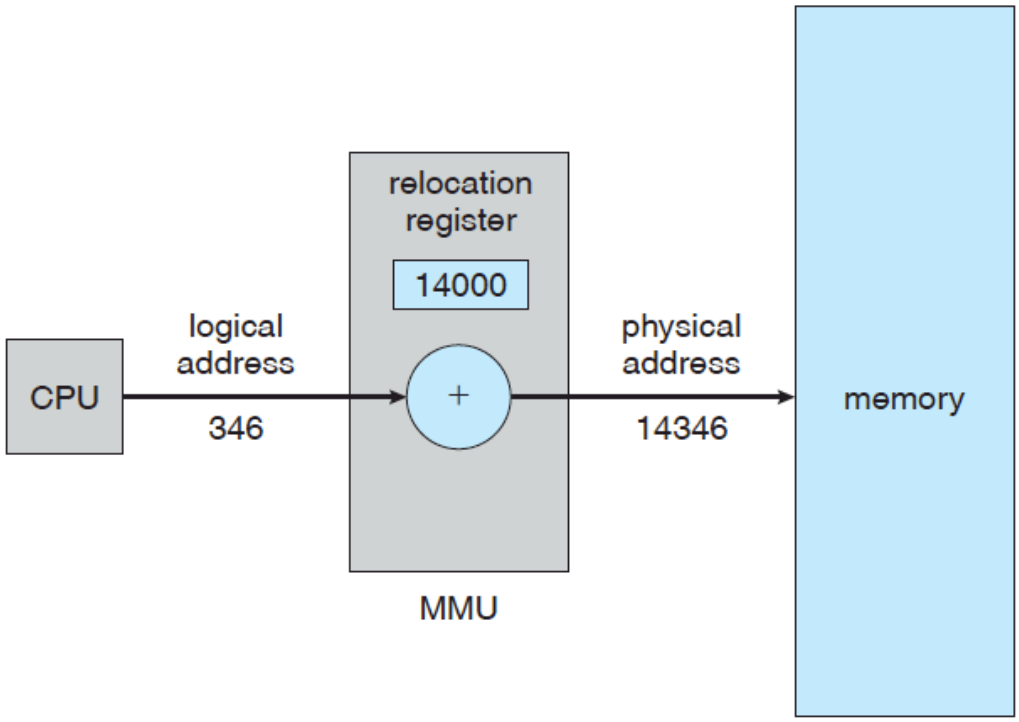
Espaços de endereçamento lógicos e físicos

Um conceito importante neste contexto é o de *compile-time binding-adress*. Um programa compilado usando o método de *binding compile-time*, gera endereços lógicos, não físicos, isto é, os endereços gerados no processo de compilação não são os endereços relativos à memória principal (endereços físicos) mas sim relativos ao inicio do programa (endereços lógicos). Ou seja, o endereço gerado é relativo à distância ao seu registo base.

Os endereços gerados pelo programa definem o espaço de endereçamento lógico. O conjunto dos endereços físicos

correspondentes ao endereços lógicos definem o espaço de endereçamento físico. Os espaços de endereçamento físicos e lógicos diferem em tempo de execução.

O mapeamento de endereços lógicos para físicos é feito através de uma componente *hardware* designada por *Memory-Management Unit (MMU).*



Se o programa tentou endereçar algo para o endereço 0, no caso da imagem acima, é imediatamente realocado para o endereço 14000. Um acesso ao endereço 346 é imediatamente realocado para o endereço 14346. O programa do utilizador nunca usa os endereços físicos efetivos.

Isto tem uma vantagem enorme, vários programas podem gerar exatamente os mesmo endereços, por exemplo, quando se faz um *fork,* mas, através deste nível de indireção dos endereços não haverá colisões sendo que cada processo irá ter o seu próprio espaço de endereçamento físico.

Com esta técnica, é também possível que o sistema operativo varie, dinamicamente, de tamanho, não afetando os outros programas pois, esses mesmos, não dependem de um endereçamento directo

para a memória, isto é, se os endereços gerados por um programa fossem estáticos, se mapeassem sempre para a mesma zona da memória central, caso o sistema operativo aumentasse de tamanho, os programas tinham que ser recompilados para obterem um novo mapeamento direto para a memória. Com o conceito de realocação, isso não é necessário pois, os programas geram endereços lógicos e, caso o sistema operativo cresça, é só preciso mudar o registo base do programa.

Em resumo, existem dois tipos de endereçamento, lógico ([0,MAX]) e físico ([R, R + MAX], onde R é um endereço base). O utilizador apenas gera endereços lógicos e pensa que o processo corre na gama de valores de 0 a *MAX.* Estes endereços têm que ser realocados antes de serem usados.

Swapping

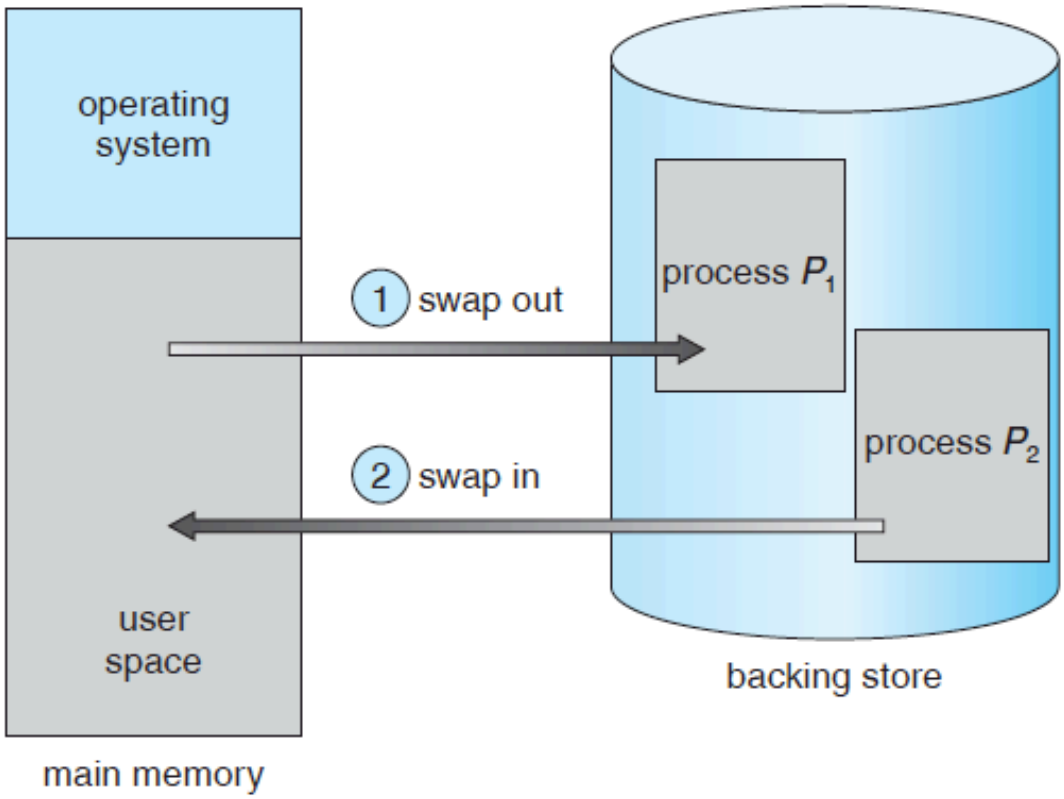
Um processo pode ser retirado da memória principal e, posteriormente, voltar a ser colocado na mesma para acabar a sua execução. Isto pode acontecer quando a memória principal não tiver mais espaço para processos. Por exemplo, imaginando o algoritmo de escalonamento baseado em prioridades, se a memória estiver cheia e aparecer um processo com maior prioridade, um processo com prioridade mais baixa poderá ser removido da memória para dar lugar a esse processo. Os processos ao saírem da memória, são guardados numa *backing store.* Um processo quando volta à memória principal, pode não ser atribuído ao mesmo espaço de endereçamento físico anterior, mas isto não será um problema porque, como já foi visto, pode-se tirar partido dos endereços lógicos para a realocação. Quando o escalonador de *CPU* decide executar um processo é chamado o *dispatcher*. O *dispatcher* verifica se o processo pretendido está na memória, e caso não esteja e esta estiver cheia, é retirado um processo da memória para dar lugar a este novo processo.

Nem todos os processos podem fazer *swap*. Imaginemos que existe um processo que está à espera de *I/O* mas a fila de espera de

um determinado dispositivo de *I/O* está muito ocupada. Se removermos o processo da memória e colocarmos lá outro, a operação de *I/O,* quando concluída, irá tentar usar a memória desse novo processo. Um processo quando é removido da memória principal tem que estar *idle.*

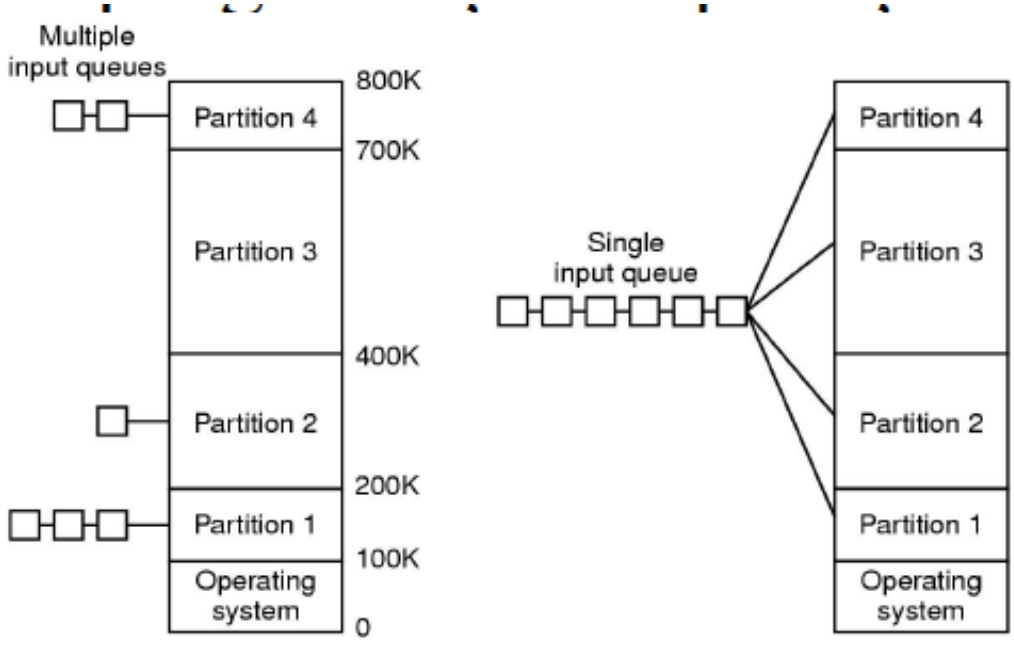
É de notar que o tempo de troca de contexto é muito elevado, sendo a maior parte do tempo perdido em transferência e, por isso, não é desejado fazer muito *swapping.* O *swapping* está normalmente desativado nos sistemas mas será ativado se muitos processos estiverem a usar muita memória principal. Será desativado outra vez quando a carga do sistema baixa.

- *Swap out* : remove um processo da memória principal.
- *Swap in* : adiciona um processo à memória principal.

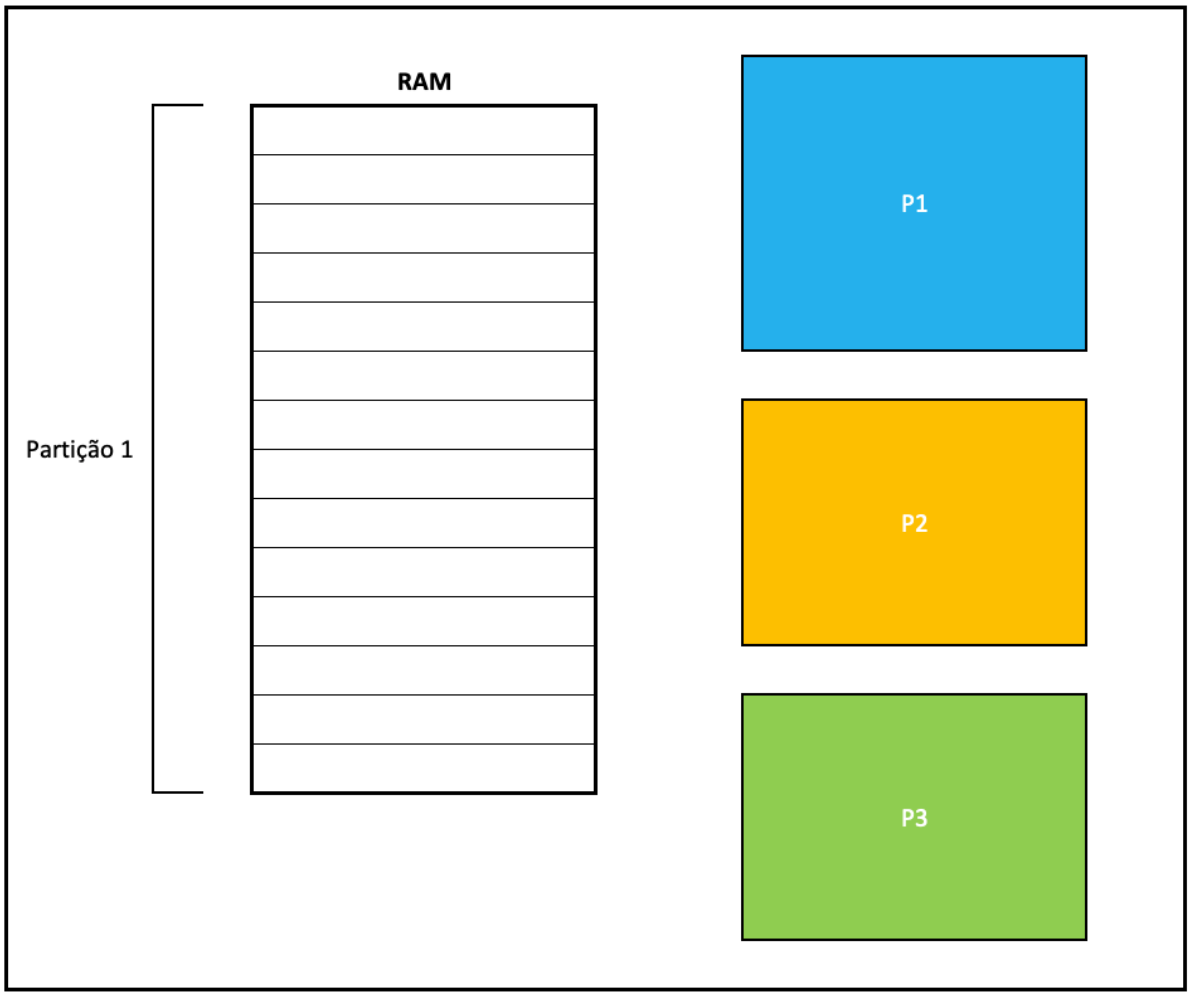


Alocação de Memória

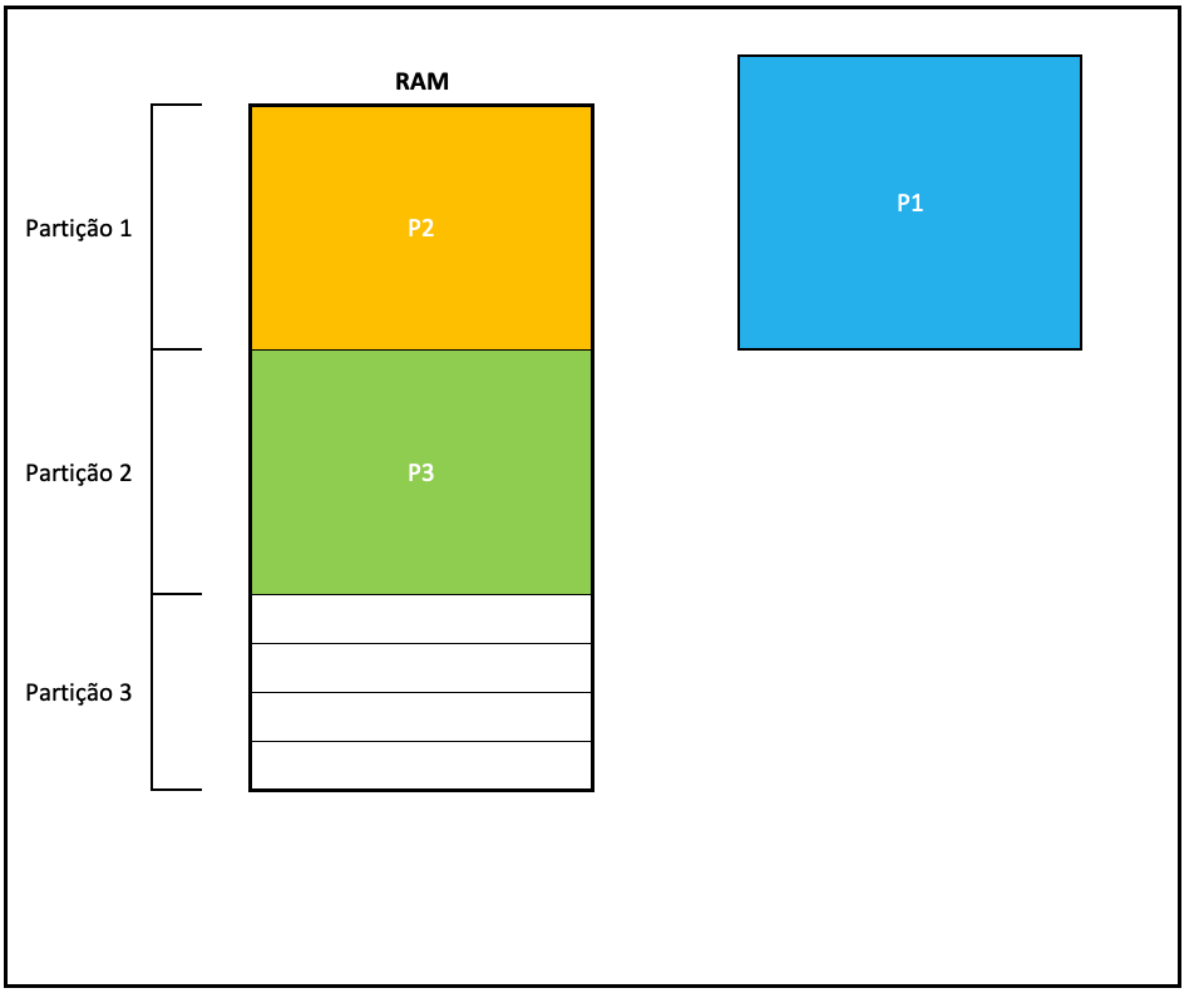
Um dos métodos simples para alocar memória é dividir a memória em partições fixas em que cada partição contém apenas um processo. Dado isto, o nível de multiprogramação, isto é, o número de processos que podem estar na memória, é limitado pelo número destas partições. Neste método, quando uma partição está livre, um processo é selecionado e colocado numa partição livre. Quando um processo termina, a partição correspondente fica livre. Este método já não é usado porque não é eficiente.



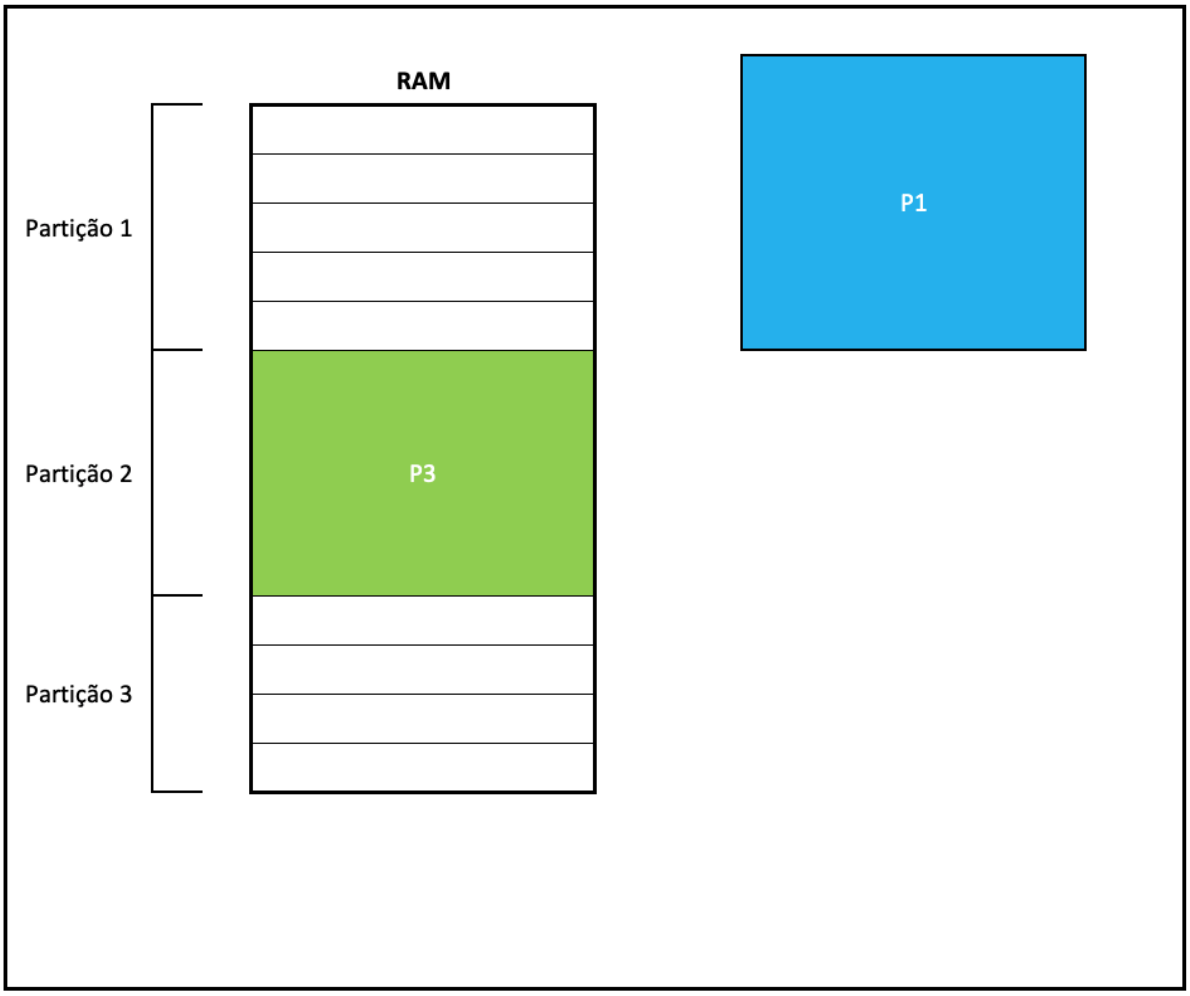
No esquema de partição variável, o sistema operativo mantém uma tabela que indica quais as partições que estão livres e as que estão ocupadas. Num estado inicial, a memória é vista como uma partição apenas, um bloco gigante de memória disponível, como se vê na imagem abaixo. O processo P1 ocupa 6 blocos, P2 5 blocos e P3 5 blocos.



A nossa RAM é composta por uma partição no estado inicial, com 14 blocos. Tem apenas uma partição porque ainda não foram alocados processos. Agora imaginemos que os processos P2 e P3 são alocados na memória. O estado desta mesma é o seguinte.



Uma tabela no sistema operativo irá conter a informação de que a partição 1 está ocupada bem como a partição 2, e que a partição 3 se encontra livre. No entanto, o tamanho dessa partição não permite que o P1 seja alocado pois não existe espaço suficiente para o alocar pois tem-se 4 blocos livres na *RAM* e o processo P1 ocupa 6 blocos. Quando o processo P2 termina a sua execução, é removido da *RAM* e o espaço de memória livre aumenta, deixando a partição 1 livre.



Embora haja agora espaço livre suficiente na memória, o processo P1 continua a não puder ser alocado pois não existe espaço contíguo na memória para o alocar. A partição 1 tem 5 blocos livres e a partição 3 tem 4 blocos livres, no entanto o processo P1 tem 6 blocos. Este problema é designado por **fragmentação de memória.**

Em síntese, a alocação de memória tem o seguinte fluxo:

- Os processos são colocados numa fila de espera para obterem *CPU.*
- O sistema operativo tem em conta os requisitos de memória de um processo e o espaço disponível na memória principal.
- Os processos vão sendo alocados na memória, nos espaços livres, formando uma partição.
- À medida que os processos terminam, deixam a partição

livre para outro processo.
- Se existirem duas partições consecutivas livres, é feito *merge* às partições.
- O sistema operativo vai escolhendo os processos que cabem nas partições, sendo que os processos que não cabem, têm de esperar, procurando na fila processos que cabem na partição.

Estratégias para atribuir uma partição a um processo:

- *First-fit :* Alocar o primeiro buraco que é largo suficiente para um processo.
- *Best-Fit :* Alocar o buraco mais pequeno que é grande suficiente para um determinado processo. É necessário percorrer a lista toda dos buracos disponíveis.
- *Worst-Fit :* Alocar o maior buraco ao processo. Mais uma vez, é necessário percorrer a lista toda de buracos disponíveis.

O ponto negativo deste tipo de alocação é o facto de os processos terem que estar contíguos na memória, o que leva à fragmentação. Uma solução seria ir juntando os blocos que estão a ser usados tornando-os contíguos, este técnica designa-se por compactação.

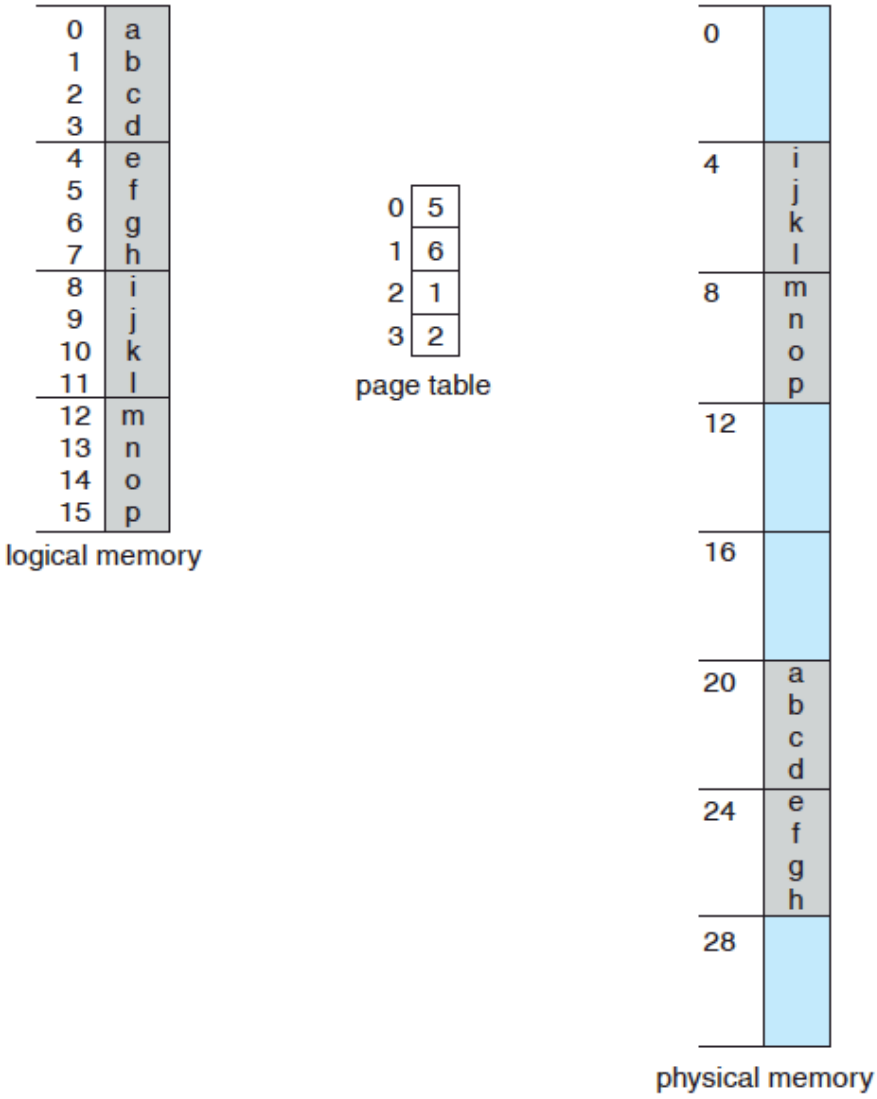
Paginação

A paginação vem resolver o problema da fragmentação de memória. A ideia da paginação é permitir que o espaço de endereçamento físico não seja contíguo, evitando a compactação e a fragmentação.

O método usado para implementar páginas é dividir a memória física em blocos de tamanho fixo, designados por *frames,* e dividir a memória lógica em blocos do mesmo tamanho, designados por páginas. A paginação tem suporte de *hardware*. Cada endereço gerado pelo *CPU* é composto por duas partes: o número da página e

o seu *offset.* O número da página é usado como índice numa tabela de paginação.

O tamanho de um bloco é normalmente uma potência de 2, pois torna a tradução de endereços mais fácil.



Assumindo que existem 4 páginas, cada uma com 4 blocos, então os endereços são mapeados da seguinte maneira:

- Endereço lógico 3 -> está contido na página 0. Vai-se

buscar o *frame* à tabela de paginação, fazendo com que o endereço lógico seja o índice da tabela, ou seja, o endereço lógico 0 é mapeado para a *Frame* 5 da memória física. Fazendo 5 * 4 (*frame* x tamanho do *frame*), somos colocados no endereço físico 20, correspondente ao *frame 5*. Somando 3 (*offset)* obtém-se o endereço físico 23. Então, o endereço físico pode ser obtido fazendo: 5 * 4 + 3 = 23.

```
pageTable[page(logicalAdress)] * pageSize + logicalAdress =
                        physicalAdress
```

A vista da memória por parte do utilizador continua a mesma, ele pensa que tem a memória para si e que contém uma zona única de memória apenas para ele. No entanto, o que acontece é que os endereços que são gerados pelo *CPU* não correspondem aos endereços reais de memória, havendo este processo de mapeamento de endereços.

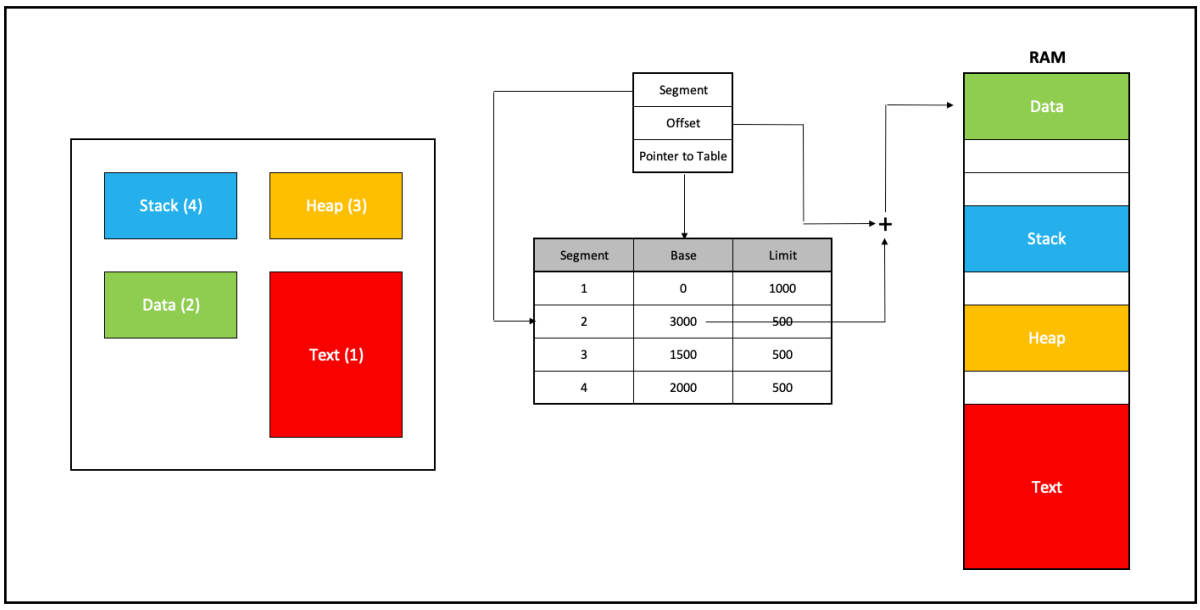
O sistema operativo contém uma *frame table* necessária para manter toda a informação das *frames* disponíveis, ocupadas, número total, *etc.*

Segmentação

Ao contrário da paginação em que o programa é dividido em páginas, todas com o mesmo tamanho, sem lógica alguma, a segmentação pretende dividir os programas de uma maneira mais lógica, uma divisão em módulos. É o que acontece com o compilador de C, em que divide o programa em:

- Código
- Variáveis globais
- *Heap*
- *Stack*
- Librarias de C

A segmentação permite que os seus módulos sejam guardados de uma maneira não contígua na memória, no entanto, os módulos em si têm que estar contíguos.



Para calcular a localização de um segmento na memória, para o exemplo acima, basta fazer os seguintes cálculos:

- Procura-se o endereço base da memória física através do número de segmento.
- De seguida, basta somar o endereço lógico do segmento e obtém-se o endereço físico.
- O sistema operativo tem que verificar se o endereço é válido, isto é, *base* + *offset* < *base* + *limit* <=> *offset* < *limit*

A segmentação apresenta o mesmo problema de alocação de memória que é a fragmentação de memória. Como se pode ver na imagem, existe 4 blocos livres mas não contíguos.

Memória Virtual

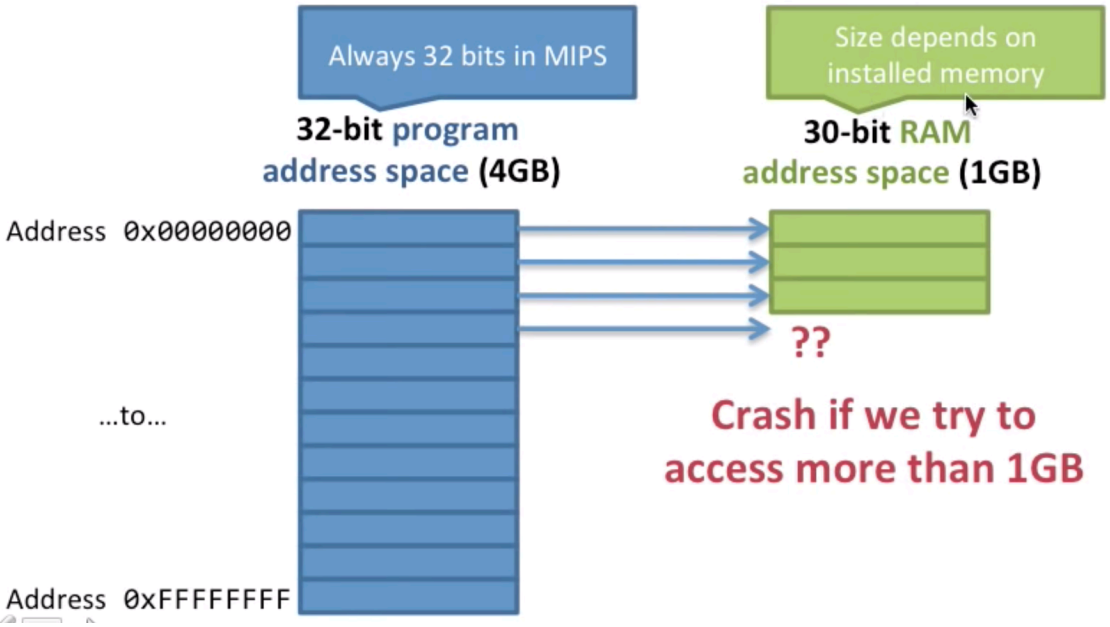
O conceito de memória virtual surge para resolver problemas de memória. Os principais problemas relacionados com a memória principal são:

- ***RAM* insuficiente**. Antigamente era impossível correr um programa cujo espaço de endereçamento fosse maior do que a memória principal. Este foi o principal motivo para a criação da memória virtual, permitir que programas maiores que a memória pudessem ser executados sem problemas.
- **Buracos no espaço de endereçamento**. Quando existem vários programas em execução pode existir espaço para novos programas na memória principal, mas como não são contíguos, novos processos podem não conseguir ser executados.
- **Programas a escreverem uns em cima dos outros**. Isto apresenta um grau de segurança muito baixo, pois os processos podem corromperem-se uns aos outros.

Problemas

Espaço Insuficiente

Se tivermos 1GB de *RAM* e um programa cujo espaço de endereçamento seja de 4GB, se o programa tenta usar mais memória do que aquela que existe na máquina, o programa vai crashar pois não consegue mapear os endereços para memória.

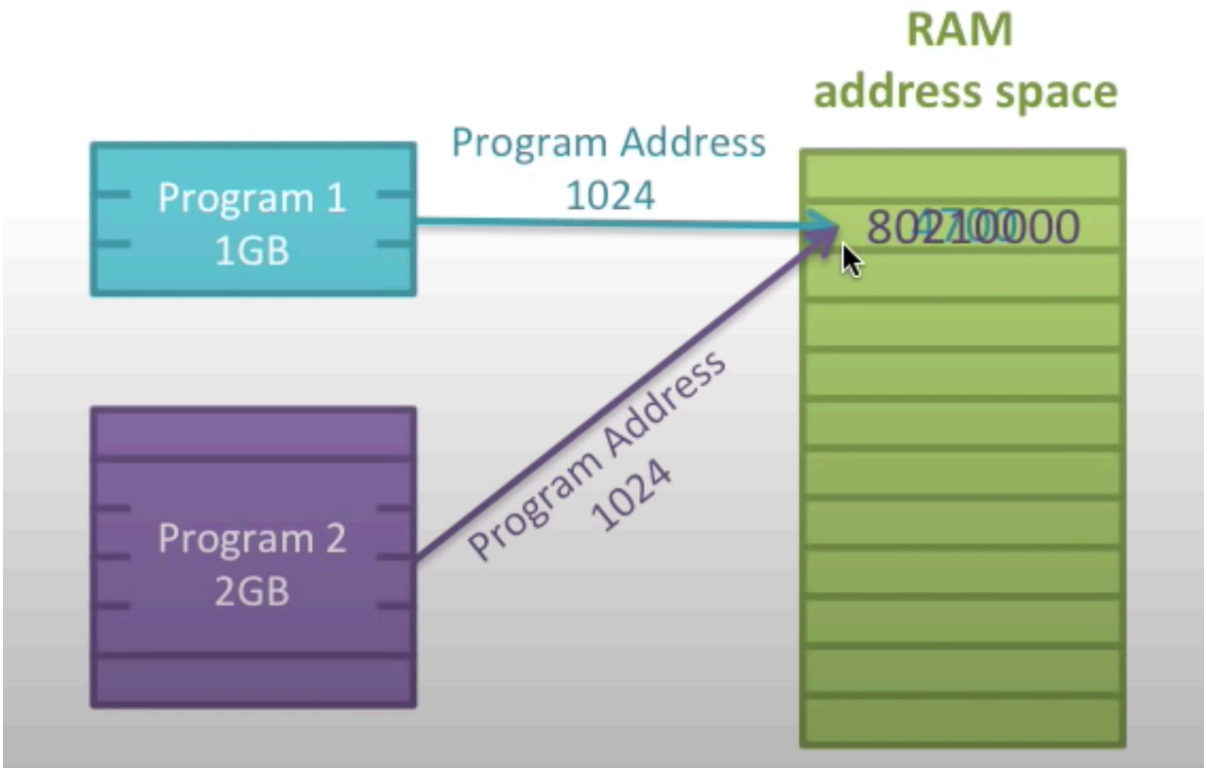


Buracos no espaço de endereçamento

Como foi mostrado na secção de alocação de memória, podem haver buracos na memória principal. Uma vez que, sem memória virtual, os programas têm que estar num espaço contíguo da memória principal, a entrada e saída de programas da mesma podem deixar buracos onde não cabem novos programas, mas que, no entanto, a soma dos espaços desses buracos permitiria alocar mais programas. Sem qualquer tipo de gestão, este acontecimento torna um sistema pouco eficiente.

Manter os programas seguros

Sem qualquer tipo de gestão, dois programas podem aceder ao mesmo espaço de endereçamento físico. Se um programa gerar o endereço 1024, e guardar nesse endereço alguma informação, outro programa que enderece algo para esse mesmo endereço poderá corromper os dados lá contidos. Sem memória virtual, não há segurança entre programas.



O problema geral de todos os exemplos mostrados, é o facto de o espaço de endereçamento dos programas serem todos o mesmo. Para resolver isto, cada programa irá conter o seu espaço de endereçamento virtual, como já foi mencionado acima. Os endereços gerados pelo *CPU* para cada programa são endereços virtuais que, posteriormente, necessitam de ser mapeados para memória física.

Solução dos problemas

Espaço insuficiente

A solução para resolver o espaço insuficiente da memória é usar o disco. Quando a *RAM* está cheia de programas e pretende alocar espaço para mais programas, remove um programa da memória para dar espaço a outro, através de uma política de escalonamento. O programa removido irá ser guardado em disco e posteriormente irá voltar a ser alocado na memória. Esta técnica é designada de

swapping e foi apresentada acima. Ao usar o disco, dá a ilusão de se ter memória ilimitada pois, mesmo que a memória esteja cheia, os programas vão sendo removidos para dar lugar a outros. No entanto, como o disco é muito lento, não se quer estar a fazer sempre *swap.*

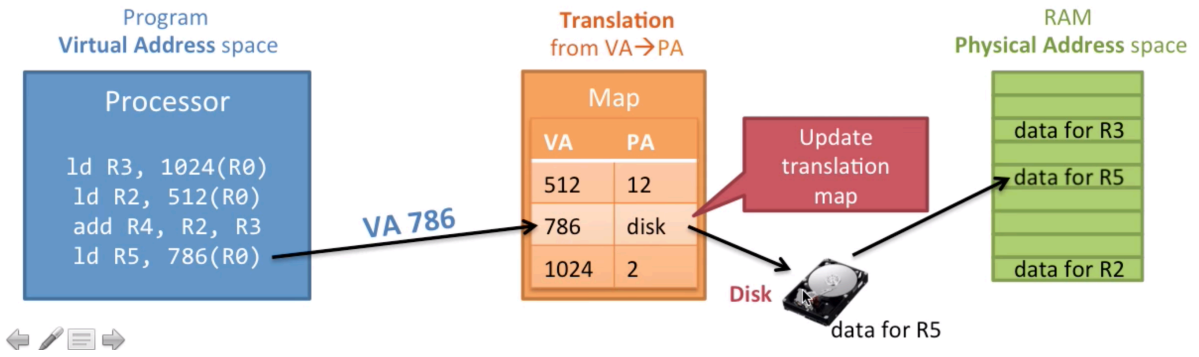
Buracos no espaço de endereçamento

Para resolver o problema dos buracos no espaço de endereçamento, as técnica de paginação permite com que isto deixe de existir pois é possível dividir um programa em vários blocos e alocá-los onde houver espaço. Os programas deixam de precisar de um espaço contíguo na memória pois, através da *MMU,* é possível gerir todas as páginas dos diferentes programas.

Manter os programas seguros

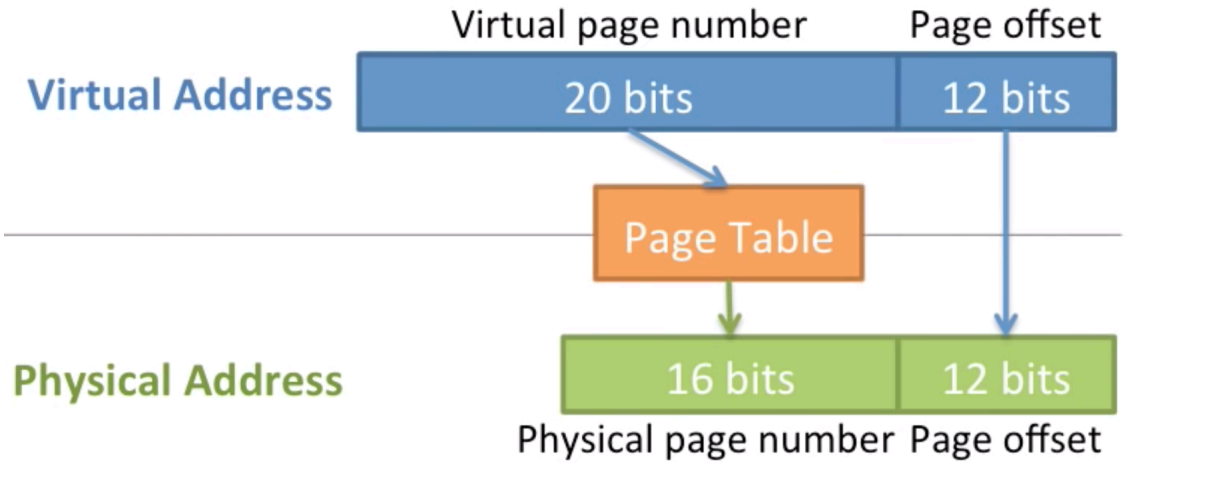
O mapeamento entre endereços lógicos e físicos permite que os programas, mesmo gerando os mesmos endereços, sejam mapeados para zonas de memória diferentes, garantindo que cada programa tem o seu espaço de endereçamento.

A indireção é a principal ferramenta da memória virtual, como se pode observar na imagem abaixo. Todos os endereços gerados de *CPU* resultantes do programa, são endereços virtuais que têm que ser traduzidos para endereços físicos. Os dados que o programa pretende aceder podem não estar em memória e necessitam de ser trazidos de disco para memória para serem acedidos, como por exemplo, acesso a ficheiros.

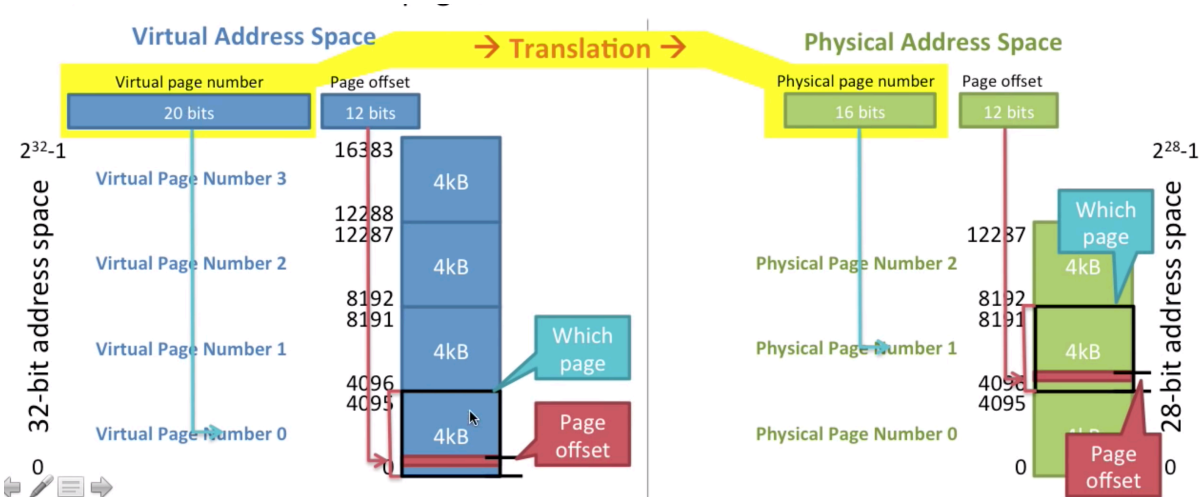


No entanto, a tabela que traduz os endereços não pode fazer o mapeamento de 1 para 1, isto é, para cada endereço virtual ter a correspondência do endereço físico. Se isso acontecesse, o espaço ocupado pela tabela era gigante. Para resolver, introduziu-se os conceitos de paginação, em que ao contrário de ter mapeamento um para um, tem mapeamento de blocos de memória maiores, normalmente de 4kB. A tabela passa a ter correspondências entre páginas de memória virtual e física.

Os endereços são então mapeados de uma maneira diferente. Um endereço virtual, neste caso, é composto por duas partes, o número da página e o *offset.* O número da página vai ser o número que vai ser usado como índice na tabela de tradução de endereços, isto é, após este cálculo sabemos em que página da memória física estará a informação que pretendemos. Mas isto só nos dá o inicio da página, não o sítio exato. Para isso é necessário somar o valor do *offset* para se ir para a localização exata.



O facto de não ser preciso fazer nenhum cálculo no valor do *offset* da página é por ter o mesmo significado nos dois lados. Significa, dado o inicio de uma página, onde está a informação pretendida. No exemplo abaixo, pode-se ver o processo de tradução de endereços. A primeira parte do endereço virtual é usado para calcular a página na memória principal, que neste caso fez a correspondência, Página 0 (memória virtual) -> Página 1 (memória física) e o *offset* permitiu ajustar o endereço para o sitio pretendido.



Como já foi mencionado anteriormente, caso uma página não esteja na memória é necessário ir buscá-la a disco. A tabela de páginas, que contém a relação entre as páginas da memória virtual e da memória física, pode referenciar que uma dada página que se procura está em disco. Quando uma página pretendida não está em memória o *CPU* gera uma excepção designada por *page fault.* Neste caso, o *hardware* salta para o *handler* de *page fault* para tratar o caso. Esta operação de *page fault* é uma operação que tem um custo demasiado elevado.

A paginação é uma das coisas que nunca queremos usar mas que é muito bom ter por perto. O lado negativo da paginação é ser muito lento o processo de ir buscar páginas a disco (*page fault)* mas, no entanto, é necessário tê-la para o computador não crashar por não ter memória suficiente. Por isso é que comprar mais memória torna o computador mais rápido, pois é menos provável ter que paginar.

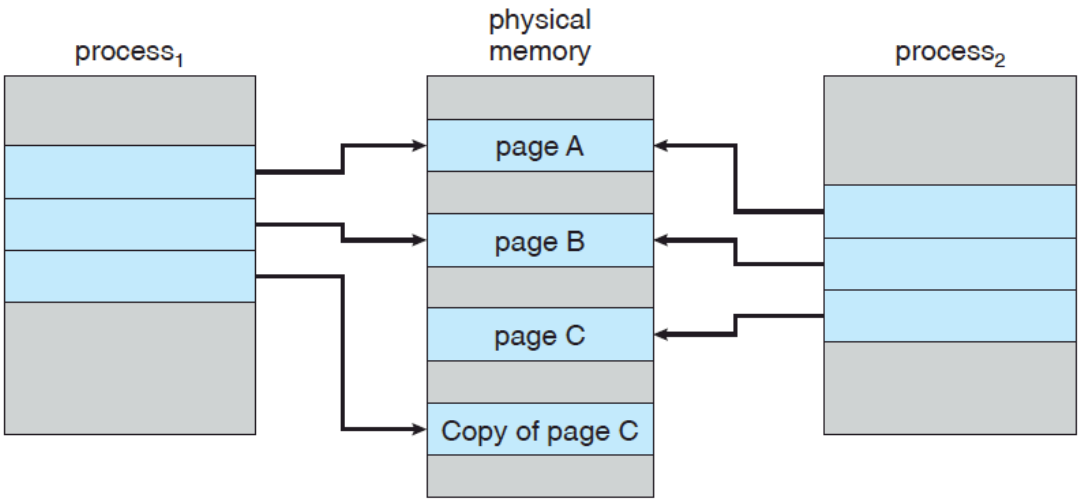
Demanding Page

Um programa, quando vai ser executado, pode ser todo carregado para a memória. O problema com esta abordagem é que poderá não ser preciso, inicialmente, carregar o programa todo para memória. Uma estratégia diferente é carregar apenas as páginas do programa à medida que estas são necessárias para a execução. As páginas nunca acedidas nunca serão carregadas para memória.

Copy-on-Write

A chamada ao sistema *fork()*, cria um processo filho que é um duplicado do seu pai. Inicialmente, quando se invocava esta chamada ao sistema, eram criadas duplicações das páginas do processo pai mas, no entanto, como a maior parte dos processo filhos invocava um *exec(),* o trabalho de duplicação era desnecessário pois a imagem do processo filho era trocado pela imagem do programa que era chamado no *exec()*. Para resolver isto foi criada a técnica *copy-on-write,* que permite que as páginas entre o processo pai e o processo filho sejam partilhadas. Estas páginas são marcadas como *copy-on-write* e, sempre que um dos processos altera algo na página, é criada uma nova página no espaço de endereçamento do processo que a alterou, mantendo as páginas que não foram alteradas partilhadas na mesma.

Como se pode ver na imagem abaixo, o processo pai e filho apontam para as mesmas páginas, sendo estas partilhadas entre os dois processos. O processo 1, ao alterar a página C, faz uma cópia da mesma e passa a estar definida no seu espaço de endereçamento.



Rejeição de páginas

A rejeição de páginas acontece quando não existem mais *frames* disponíveis na memória para serem utilizadas. Neste caso, o sistema operativo terá que remover uma para dar lugar a outra. O fluxo desta operação é o seguinte:

1. Encontrar a página em disco pretendida.
2. Encontrar uma *frame* livre na memória.
    1. Se existe, usá-la.
    2. Se não existe, usa-se um algoritmo de rejeição de páginas para selecionar a “vitima”, isto é, a página que vai ser removida.
    3. Escrever a página da “vítima” em disco, mudando as tabelas de acordo com esta operação.
3. Carregar a nova página para a *frame* que agora esta livre, mudando as tabelas de paginação e de *frames*.
4. Restaurar o processo do utilizador.

É de notar que este algoritmo requere duas *page faults,* uma para remover a página de memória e outra para carregar a nova página para a mesma, o que torna isto muito ineficiente. No entanto, existe apoio no *hardware* para permitir que este processo seja mais

eficiente. Existe um *bit* nas tabelas (*dirty bit)* que representa o facto de uma página ter sido modificada ou não. Se uma página não foi modificada, não é necessário fazer cópia para disco, pois a imagem da página do disco é a mesma. No entanto, se foi alterada, então é necessário escrevê-la em disco.

Existem vários algoritmos de rejeição de páginas, com o objectivo de seleccionar uma página para sair de memória para dar lugar a outra.

- *FIFO Page Replacement* : associada a cada página, o tempo em que foi trazida para memória. Quando à rejeição de páginas, a página que está à mais tempo no sistema é removida da memória. Poderá não ser necessário usar o tempo, e, ema alternativa, usar uma *FIFO queue,* onde a página que está à cabeça é a que vai sair da memória e, sempre que entra uma página em memória é colocada no fim da lista. A desvantagem deste algoritmo é o facto de a página que está à mais tempo em memória ser a mais usada pelo sistema.
- *LRU Page Replacement* : LRU = Least-Recently-Used. É rejeitada a página que não está a ser usada à mais tempo. Associa a cada página o tempo da última vez usada. Assume que as páginas usadas recentemente são as mais usadas. Mantém uma fila com as páginas ordenadas com este critério.
- *Second-Change Replacement* : funciona como o algoritmo *FIFO Page Replacement* mas, se a página mais antiga, a que está à cabeça da lista, foi usada recentemente, não a remove e passa à próxima página da lista, dando uma segunda oportunidade à mesma.
- *NRU Page Replacement* : Cada página tem 1 *bit* de acesso e 1 de escrita. As páginas têm uma classificação: (1) Não acedida & Não modificada (2) Não acedida & Modificada (3) Acedida & Não modificada (4) Acedida & Modificada. O algoritmo remove a página com menor *ranking.*

Alocação global vs. Alocação Local

- Alocação global : um processo pode escolher um *frame* para ser rejeitado do conjunto de todos os *frames* disponíveis na memória, mesmo que esse *frame* possa pertencer a outro processo. Ou seja, um processo pode tirar o *frame* de outro.
- Alocação local : um processo só pode escolher dentro do seu conjunto de *frames* disponíveis, um *frame* para ser rejeitado.

Trashing

O processo está a fazer *trashing* se está a passar mais tempo em paginação do que em execução. Imaginemos um processo que precisa de um número considerável de *frames* para ser executada mas existem poucas disponíveis, o processo irá gerar *page faults* muito rapidamente. Neste momento, terá que trocar de páginas, enviando uma página da memória para o disco mas, que no entanto, essa página removida será necessária num curto intervalo de tempo e então, consequentemente, irá gerar outra *page fault.*

Tamanho das páginas

Vantagens de páginas pequenas:

- Menos fragmentação interna.
- Melhor adequação a várias estruturas de dados e código.
- Menos partes de programas não usados em memória
- I/O reduzido

Desvantagens de páginas pequenas:

- Tabelas de paginação maiores.
- Mais páginas