Resumo Prova 2 SO

Guilherme Christopher Michaelsen Cardoso *

21 de junho de 2017

*baseado no livrão do Tanenbaum, feito em LaTeX

1 Resumão de SO (Prova 2)

1.1 Espaços de endereçamento

Expor memória física para processos tem diveras desvantagens:

- Permitir que programas de usuário enderecem qualquer byte de memória torna possível quebrar o sistema operacional.
- Com esse modelo, é dificil ter vários programas rodando ao mesmo tempo.

1.1.1 Noção de espaço de endereçamento

É necessário resolver dois problemas antes de permitir múltiplas aplicações em memória ao mesmo tempo: proteção e relocação.

- Espaço de Endereçamento:
  - Conjunto de endereços que um processo pode usar para endereçar memória.
  - Cada processo tem seu próprio espaço de endereçamento, independente dos outros processos (exceto em circunstâncias especiais onde processos querem compartilhar seus espaços de endereçamento).

1.2 Memória Virtual

- Enquanto a capacidade das memórias cresce rapidamente, o tamanho dos programas cresce ainda mais rápido.
- Necessidade de rodar programas que são grandes demais para caber em memória
- Swapping não é uma alternativa desejável, devido à lentidão dos discos rígidos.
- **Memória virtual** foi a solução encontrada para esse problema.
  - Cada programa tem seu próprio espaço de endereçamento, que é dividido em partes chamadas **páginas**.
  - Cada página é um intervalo contíguo de endereços, que são mapeados em memória física.
  - Nem todas as páginas precisam estar em memória física ao mesmo tempo para rodar o programa.
  - Quando o programa referencia uma parte do endereço que já está em memória física, o hardware realiza o mapeamento necessário na hora.
  - Quando o programa referencia uma parte do espaço de endereçamento que não está em memória, o SO recebe um aviso para buscar a página faltante e reexecutar a instruçao que falhou.

  - Funciona muito bem em sistemas multiprogramados com pedaços de vários programas na memória ao mesmo tempo. Quando um programa está esperando uma página ser buscada do disco, a CPU pode ser concedida a outro processo.

1.2.1 Paginação

- A maioria dos sistemas de memória virtual usam **paginação**
- Programas geram **endereços virtuais** de memória, formando o **espaço de endereçamento virtual**.
- Em sistemas com memória virtual, esses endereços não são enviados diretamente ao barramento da memória. Ao invés disso, vão para uma **MMU (Memory Management Unit)** que mapeia o endereço virtual em endereços de memória reais.
- O espaço de endereçamento virtual consiste de unidades de tamanho fixo chamadas **páginas**. AS unidades correspondentes em memória física são chamadas de **molduras de página**. As páginas e molduras de página geralmente tem o mesmo tamanho.
- Supondo um sistema de memória com páginas de 4KB, 64KB de espaço de endereçamento virtual e 32KB de memória física, temos 16 páginas virtuais e 8 molduras de páginas.
- Sempre que se necessita buscar um item no disco, é necessário que a página inteira seja buscada.
- Vários processadores suportam múltiplos tamanhos de página que podem ser misturados pelo SO de acordo com a necessidade.
  - Ex.: A arquitetura x86_64 suporta páginas de 4KB, 2MB e 1GB, sendo possível, por exemplo, utilizar páginas de 4KB para aplicações de usuário e uma única página de 1GB para o kernel do SO.
- Exemplo: Supondo páginas de 4KB, 64KB de espaço de endereçamento virtual e 32KB de memória física. Um programa solicita um dado que está no endereço virtual 0. Supondo que a página que contém os endereços 0 a 4095 está mapeada na moldura de página numero 2. Qual o endereço de memória real que será emitido pela MMU?
  - A moldura 0 contém os endereços 0 a 4095. A moldura 1 contém os endereços 4096 à 8191. Logo, a moldura 2 começa no endereço 8192 e vai até 12287. Portanto, o endereço físico que será emitido será 8192.
- Exemplo 2: O endereço virtual 20500 é 20 bytes à partir do começo da página virtual 5 (endereços virtuais 20480 a 24575). Supondo que essa página seja mapeada para a moldura de página número 3 (12k-16k), o endereço físico emitido pela MMU será 12288 + 20 = 12308.
- Como existem mais páginas virtuais do que moldura física, é necessário controlar quais páginas virtuais estão presentes em memória física. Para isso, é usado um bit de controle (**bit presente/ausente**).

- Caso o programa referencie um endereço não mapeado, a MMU detecta isso e faz com que a CPU interrompa o sistema operacional. Essa interrupção é chamada de **page fault**. O SO então escolhe uma moldura de memória pouco usada, salva seu conteúdo no disco (caso já não esteja lá), e então busca (também do disco) a página que acabou de ser referenciada, colocando-a na moldura de página que foi liberada, refaz o mapeamento, e reinicia a instrução interrompida.
- Exemplo: supondo um endereço virtual 8196 (0010000000000100 em binário), sendo mapeado pela MMU. Esse endereço virtual de 16 bits é dividido em duas partes: um número de página (4 bits) e um offset (12 bits). Com 4 bits de offset, podemos endereçar $2^4 = 16$ páginas, e com um offset de 12 bits, podemos endereçar todos os $2^{12} = 4096$bytes em uma página.
- O número da página é usado como índice na **tabela de páginas**, representando o número da moldura de página correspondente àquela página virtual. Se o bit *presente/ausente* é 0, uma interrupção ao SO é causada. Se é 1, o número da moldura de página encontrada na tabela de páginas é copiado para os 3 bits mais significativos do registrador destino, junto com o offset de 12 bits, que é copiado do endereço virtual. Juntos, eles formam um endereço físico de 15 bits. Esse endereço é então enviado ao barramento da memória como o endereço físico de memória.

1.2.2 Tabelas de página

Em uma implementação simples, o mapeamento de endereços virtuais em endereços físicos pode ser feito da seguinte forma:

- O endereço virtual é dividido em um número de página virtual (bits mais significativos) e um offset (bits menos significativos)
  - Por exemplo, em um endereço de 16 bits com páginas de 4KB, os 4 bits mais significativos especificam uma das 16 páginas virtuais e os 12 bits menos significativos representam o byte offset dentro da página selecionada.
  - É possível usar endereços menores ou maiores para indexar a página, definindo tamanhos diferentes de páginas.
- O número de página virtual é usado como índice na tabela de páginas para encontrar a entrada para essa página virtual. A partir da entrada na tabela de páginas, o número da moldura de página (se existir), é encontrado.
- O número da moldura de página é concatenado aos bits mais significativos do offset, substituindo o número da página virtual, para formar um endereço físico que pode ser enviado para a memória.

Portanto, o propósito da tabela de páginas é mapear páginas virutais em molduras de página.

1.2.3 Estrutura de uma entrada na tabela de páginas

O layout exato de uma entrada na tabela de páginas é altamente dependente da máquina, mas a informaçao presente normalmente é a mesma.

- O tamanho varia de computador a computador, mas 32 bits é um tamanho comum.
- O campo mais importante é o o *número da moldura de página.*
- Em seguida, existe o bit *presente/ausente.* Se esse bit for 1, a entrada é válida e pode ser usada, caso contrário, a página virtual não está atualmente em memória. Acessar uma entrada na tabela de páginas que tenha esse bit em 0 causa um page fault.
- Os bits de proteção dizem quais tipos de acesso são permitidos. Na forma mais simples, se tem apenas um bit, que permite apenas leitura se estiver em 0, e leitura ou escrita se estiver em 1. Uma alternativa é usar 3 bits, um para leitura, um para escrita e um para execução.
- Os bits *modificado* e *referenciado* controlam a utilização da página. Quando ocorre uma escrita em uma página, o hardware seta o bit *modificado.* Esse bit é importante quando o SO decide usar aquela moldura de página. Se a página tiver sido modificada (suja), ela deve ser escrita novamente no disco. Se não, ela pode ser abandonada, já que a cópia em disco ainda é válida. Esse bit normalmente é chamado de **dirty bit**.
- O bit *referenciado* é setado toda vez que uma página for referenciada, tanto para leitura quanto para escrita. Seu valor é usado para ajudar o SO a escolher uma página quando ocorre um page fault. Páginas que não estão sendo usadas são melhores candidatas a serem substituídas.
- O último bit permite desabilitar o caching na página. Isso é importante para páginas que mapeiam direto em registradores de dispositivos ao invés de memória. Se o sistema operacional estiver em loop esperando um dispositivo de I/O responder um comando, ele precisa buscar constantemente a word do dispositivo, e não usar uma cópia em cache desatualizada. Máquinas que não utilizam I/O mapeado em memória não precisam deste bit.

1.3 Acelerando a Paginação

Em qualquer sistema de paginação, dois problemas devem ser enfrentados:

1. O mapeamento do endereço virtual para o endereço físico deve ser rápido.
2. Se o endereço virtual for grande, a tabela de páginas também será.

O primeiro problema ocorre devido à natureza dos programas. A cada instrução executada, é necessário fazer mapeamento de endereços virtuais para físicos, já que todas as instruções são buscadas em memória e muitas delas também referenciam operandos em memória.

O segundo problema ocorre porque todos so computadores modernos utilizam endereços virtuais de 32 ou 64 bits. Usando páginas de 4KB, um endereço de 32 bits é capaz de referenciar 1 milhão de páginas, e um endereço de 64 bits referencia um número absurdamente grande de páginas. Com 1 milhão de páginas no espaço de endereçamento virtual, a tabela de páginas deve ter 1 milhão de entradas, e cada processo precisa da sua própria tabela de páginas (por possuir seu próprio espaço de endereçamento).

1.3.1 Translation Lookaside Buffers (TLB)

A maioria das técnicas de otimização de paginamento parte do princípio que a tabela de páginas está em memória. Se uma instrução copia um registrador para outro, por exemplo, em um sistema sem paginação, ela faz apenas uma referência à memória (para buscar a instrução em si). Com paginamento, pelo menos uma referência a mais é necessária, para acessar a tabela de páginas. Devido ao impacto dos acessos de memória na performance da CPU, dobrar o número de acessos à memória diminui potencialmente pela metade o desempenho.

Baseado na observação de que a maioria dos programas tende a fazer um grande número de referências a um número pequeno de páginas, foi-se possível desenvolver uma solução para esse problema.

- Essa solução consiste em equipar computadores com um componente em hardware que é capaz de mapear endereços virtuais em endereços físicos sem precisar acessar a tabela de páginas. Esse dispositivo é chamado de **TLB** (Translation Lookaside Buffer).
- Esse dispositivo normalmente faz parte da MMU e consiste de uma tabela com um pequeno número de entradas, por exemplo, 8 (raramente mais do que 256). Cada entrada contém informação sobre uma página, incluindo o número da página virtual, um bit que indica se a página foi modificada, o código de proteção (permissões read/write/execute), e o número da moldura de página física na qual a página está localizada. Esses campos possuem correspondência 1 para 1 com os campos na tabela de páginas, exceto pelo número da página virtual, que não é usado pela tabela de páginas. Outro bit indica se a entrada é válida ou não.
- Quando um endereço virtual é enviado à MMU para tradução, o hardware primeiro checa para ver se é um número de página virtual presente na TLB comparando esse endereço paralelamente com todas as entradas na TLB. Se um cnadidato válido é encontrado e o acesso não viola os bits de proteção, a moldura de página é buscada diretamente do TLB, sem necessitar acessar a tabela de páginas.
- Se o número de página virtual está presente na TLB, mas a instrução está tentando escrever em uma página protegida contra escrita, uma protection fault é gerada.
- Quando um número de página virtual não está no TLB, a MMU detecta o miss e faz uma busca na tabela de páginas. Ela então remove uma das entradas do TLB e substitui com a entrada na tabela de páginas que acabou de encontrar. Se essa página for usada novamente em breve, ela será encontrada no TLB.

- Quando uma entrada é removida do TLB, o bit de modificado é copiado de volta para a entrada da tabela de páginas em memória. Os outros valores já estão lá, exceto o bit de referência. Quando o TLB é carregado da tabela de páginas, todos os campos são buscados em memória.

1.3.2 Gerenciamento de TLB por software

Até agora, assumiu-se que todas as máquinas com memória virtual paginada possuem tabelas de páginareconhecidas pelo hardware, mais um TLB. Nesse tipo de projeto, o gerenciamento do TLB e das faltas de TLB são feitos inteiramente pela MMU. Traps para o SO ocorrem apenas quando uma página não está presente em memória.

Porém, muitas máquinas RISC fazem todo o gerenciamento de páginas em software. Nessas máquinas, entradas na TLB são carregadas explicitamente pelo SO.

- Quando um miss no TLB ocorre, ao invés da MMU buscar a referência necessária na tabela de páginas, ele apenas gera uma faulta de TLB e lança o problema para o SO. O sistema então deve encontrar a página, remover uma entrada do TLB, inserir a nova entrada, e reiniciar a instrução faltante. Como misses de TLB são mais comuns do que page faults, esse processo deve ser rápido.
- Se o TLB for razoavelmente grande para reduzir a miss rate, o gerenciamento por software de TLB se torna aceitavelmente eficiente.
- Vantagem: MMU muito mais simples, liberando espaço no chip da CPU para caches e outros componentes que possam melhorar a performance.

Várias estratégias foram desenvolvidas para melhorar o desempenho em máquinas que gerenciam TLB por software.

- Uma forma é buscar reduzir tanto os misses de TLB quanto o custo de um miss na TLB quando eles ocorrem.
- Para reduzir misses no TLB, o sistema operacional pode usar sua intuição para descobrir quais páginas tendem mais a serem usadas logo e carregar suas entradas na TLB.
- A forma normal de processar um miss de TLB, tanto em hardware quanto em software, é buscar na tabela de páginas a página referenciada. O problema de se fazer essa busca em software é que as páginas que contém a tabela de páginas podem não estar no TLB, o que causa faltas adicionais no TLB durante o processamento. Essas faltas podem ser reduzidas mantendo um grande cache em software de entradas na TLB em uma localização fixa cuja página sempre seja mantida na TLB. Ao checar primeiro o cache em sofware, o SO pode reduzir substancialmente as faltas na TLB.

Ao usar gerenciamento de TLB por software, é essencial entender a diferença entre diferentes tipos de misses. Um **soft miss** ocorre quando a página referenciada não está na TLB,

mas está em memória. Messe caso, tudo que é necessário é atualizar o TLB, sem necessidade de I/O no disco. Tipicamente, um soft miss leva de 10 a 20 instruções para ser tratado e pode ser completado em alguns nanosegundos. Em contraste, um **hard miss** ocorre quando a página em si não está em memória e nem na TLB. Um acesso a disco é necessário para buscar a página, que pode levar vários milisegundos, dependendo do disco sendo usado. Um hard miss é mais de 1 milhão de vezes mais lento do que um soft miss. O processo de busca na hierarquia da tabela de páginas é chamado de **page table walk**.

Os misses não são apenas soft ou hard. Por exemplo, se o page walk não encontrar o a página na tabela de páginas de um processo, resultando em uma page fault. Existem 3 possiblidades.

1. A página pode estar em memória, mas não na tabela de páginas do processo. Por exemplo, pode ter sido trazida do disco por outro processo. Nesse caso, não é necessário acessar o disco novamente, bastando mapear a página apropriadamente nas tabelas de página. Esse tipo de soft miss'e chamado de **minor page fault.**
2. A página precisa ser trazida do disco. Este é o **major page fault**.
3. O programa simplesmente acessou um endereço inválido e não é necessário realizar nenhum tipo de mapeamento no TLB. Nesse caso, o SO normalmente mata o programa com um **segmentation fault**. Este é o único caso em que o programa fez algo de errado. Os outros casos são automaticamente tratados pelo hardare e/ou pelo SO.

1.4 Tabelas de página para memórias grandes

1.4.1 Tabelas de página multinível

- Solução para lidar com espaços de endereçamento virtuais muito grandes.
- Assumindo um endereço virtual de 32 bits que é particionado em um campo PT1 de 10 bits, um campo PT2 de 10 bits e um campo Offset de 12 bits. Como offsets são de 12 bits, cada página tem 4KB, e existe um total de 2^{20} delas.
- O segredo para o método de tabela de páginas multinível está em evitar manter todas as tabelas de páginas em memória o tempo todo. As que não são necessárias, não devem ser mantidas.
- O campo PT1 de 10 bits endereça a tabela de página top-level, com 1024 entradas. Quando um endereço virtual é enviado para a MMU, ele extrai o campo PT1 e usa ese valor como um índice na tabela de páginas top-level. Cada uma dessas 1024 entradas na tabela de página representa 4M porque o espaço de endereçamento vitual de 4GB (32bits) foi dividido em fatias de 4096 bytes.
- A entrada encontrada indexando a tabela de páginas top-level contém o endereço ou o número da moldura de página de uma tabela de páginas second-level. A entrada 0 da tabela top-level aponta para a tabela de páginas da área de texto do programa, a entrada

1 para a tabela de páginas da área de dados, e a entrada 1023 para a tabela de páginas da pilha. As outras entradas não são usadas.

- O campo PT2 agora é usado como um índice na tabela de páginas de segundo nível selecionada para encontrar o número da moldura de página da página em si.

- Exemplo: Considere o endereço virtual de 32 bits 0x00403004, que se refere a 12,292 bytes dentro do segmento de dados. Esse endereço virtual corresponde a PT1 = 1, PT2 = 3 e Offset = 4. A MMU usa PT1 para indexar a tabela top-level, e obter a entrada 1, que corresponde aos endereços 4M a 8M -1. Ele então usa a PT2 para indexar a tabela second-level que acabou de encontrar e extrair a entrada 3, que corresponde aos endereços 12288 a 16383 dentro dessa fatia de 4M. Essa entrada contém o número da moldura de página contendo o endereço virtual 0x00403004. Se essa página não está presente em memória, o bit *presente/ausente* na tabela de páginas terá o valor zero, causando um page fault. Se a página estiver presente em memória, o número da moldura de página tirada da tabela de páginas second-level é combinado com o offset(4) para construir o endereço físico. Esse endereço é enviado ao barramento da memória.

- É interessante notar que apesar de o espaço de endereçamento conter mais de um milhão de páginas, apenas quatro tabelas de páginas são necessárias: a tabela top level, e as tabelas second-level de 0 a 4M (para o segmento de texto), 4M a 8M (para o segmento de dados) e os ultimos 4M (para a pilha). Os bits *presente/ausente* do resto das 1021 entradas da tabela de página top-leval são setados para 0, forçando um page fault se eles forem acessados. Se isso ocorrer, o sistema operacional vai notificar que o processo está tentando referenciar memória que não deveria referenciar, e tomar uma ação apropriada, como enviar um sinal para matar o processo.

- Esse sistema de dois níveis de tabelas de páginas pode ser expandido para um sistema de três, quatro ou mais níveis. Mais níveis provêm mais flexibilidade. O processador Intel 80386, por exemplo, era capaz de endereçar até 4GB de memória usando uma tabela de dois níveis que consistia de um **diretório de páginas** cujas entradas apontavam para a tabela de páginas, que, por sua vez apontava para as molduras de página de 4KB. Tanto o diretório de páginas quanto as tabelas de página continham 1024 entradas, dando um total de $2^{10} * 2^{12} = 2^{32}$ bytes endereçáveis, como desejado.

- 10 anos depois, o Pentium Pro introduziu outro nível: **tabela de ponteiros do diretório de páginas**. Além disso, extendeu cada entrada em cada nível da hierarquia da tabela de páginas de 32 para 64 bits, de forma a tornar possível acessar memória acima do limite de 4GB. Como só possuía 4 entradas na tabela de ponteiros do diretório de páginas, 512 em cada diretório de páginas e 512 em cada tabela de páginas, o total de memória que conseguia acessar ainda estava limitado a um máximo de 4GB.

- Quando um suporte apropriado de 64 bits foi adicionado à familia x86 (originalmente pela AMD), foi adicionado mais um nível, o **page map level 4**. Todas as tabelas agora possuem 512 entradas, tornando possível endereçar $2^9 * 2^9 * 2^9 * 2^9 * 2^{12} = 2^{48}$ bytes. Ainda era possível adicionar mais um nível, mas provavelmente se atingiu a conclusão de que 256TB são suficientes por um bom tempo.

1.5 Algoritmos de substituição de páginas

Quando um page fault ocorre, o sistema operacional precisa escolher uma página para remover da memória, liberando espaço para outras páginas. Se a página que será removida foi modificada enquanto estava em memória, ela deve ser reescrita no disco, para atualizar a cópia em disco. Se a página não foi alterada, não é necessário reescrever nada. A página a ser lida apenas substitui a página removida.

1.5.1 Algoritmo NRU: Not Recently Used

- A maioria dos computadores com memória virtual possuem dois bits de status: R e M, associados a cada página. R é setado toda vez que a página é referenciada (lida, ou escrita). M é setado toda vez que ocorre uma escrita na página (isto é, ela é modificada). Os bits são contidos em cada entrada na tabela de páginas. É importante que esses bits sejam atualizados em todas as referências à memória, para que eles sejam setados pelo hardware. Quando um bit é setado para 1, ele permanece em 1 até o SO resetá-lo.
- Se o hardware não tiver esses bits, eles podem ser simulados usando os mecanismos de page fault e de clock interrupt do sistema operacional. Quando um processo é iniciado, todas as suas tabelas de página são marcadas como não estando em memória. Assim que qualquer página for referencia, um page fault ocorrerá. O sistema operacional então seta o bit R (nas suas tabelas internas), muda a entrada da tabela de páginas para apontar para a página correta em modo READ ONLY, e reinicia a instrução. Se a página for subsequentemente modificada, outro page fault vai ocorrer, permitindo que o sistema operacional sete o M bit e mude o modo da página para READ/WRITE.
- Os bits R e M podem ser usados para implementar um algoritmo de paginação simples.
  - Quando um processo é iniciado, ambos os bits para todas as páginas são setados para 0 pelo SO. Periódicamente (e.g., em cada interrupção de clock), o bit R é zerado, para distinguir páginas que não foram referenciadas recentemente das que foram referenciadas.
  - Quando um page fault ocorre, o SO inspeciona todas as páginas e as divide em quatro categorias, baseadas nos valores atuais de seus bits R e M:
    - Classe 0: não referenciado, não modificado.
    - Classe 1: não referenciado, modificado.
    - Classe 2: referenciado, não modificado.
    - Classe 3: referenciado, modificado.
  - O algoritmo **NRU (Not Recently Used)** remove uma página aleatória da classe não-vazia de menor número. Implícita neste algoritmo está a idéia de que é melhor remover uma página modificada que não tenha sido referenciada em pelo menos um ciclo de clock, do que uma página limpa que está sendo muito usada. A principal vantagem do NRU é que ele é simples de entender, moderadamente eficiente de implementar e provê uma performance que, apesar de não ser ótima, pode ser adequada.

1.5.2 O algoritmo First In, First Out (FIFO)

- Outro algoritmo de paginação de baixo overhead é o **FIFO (First-In, First-Out**.
- O sistema operacional mantem uma lista de todas as páginas atualmente em memória, seguindo a ordem de chegada (a mais recente no fim da fila e a mais antiga no começo da fila). Em caso de page fault, a página no começo da fila (isto é, a que está na fila a mais tempo) é removida, e a nova página é inserida no fim da fila.
- O problema deste algoritmo está no fato de que a primeira página da fila sempre será removida, mesmo que ela seja frequentemente acessada. Por isso, FIFO em sua forma pura raramente é utilizado.

1.5.3 Algoritmo da Segunda Chance

- Uma modificação simples de FIFO que evita o problema de descartar uma página que esteja sendo muito usada consiste em monitorar o R bit da página mais antiga. Se o bit for 0, a página é antiga e não foi usada recentemente, então é substituída imediatamente. Se o bit R for 1, esse bit é resetado e a página é colocada no fim da fila, e seu tempo de carga é atualizado como se ela tivesse acabado de ser colocada em memória. Então, a busca continua.
- Esse algoritmo é chamado de **segunda chance**.
- O algoritmo procura em todas as páginas da fila até encontrar uma página que esteja no começo da fila e que não tenha sido utilizada recentemente (R = 0).
- Caso R seja 1, além de inserir a página no fim da fila, o bit é resetado. Isso garante que essa página não receba uma terceira chance, isso é, caso nenhuma página com R = 0 seja encontrado, o algoritmo funcionará como uma FIFO normal, garantindo que o algoritmo sempre termine.

1.5.4 Algoritmo do relógio

- Apesar do algoritmo da segunda chance ser razoável, ele é desnecessariamente ineficiente, pois está constantimente movendo páginas dentro da lista.
- Uma abordagem melhor consiste em manter todas as molduras de página em uma lista circular, em forma de relógio. O ponteiro do relógio aponta para a página mais antiga.
- Quando um page fault ocorre, a página para a qual o ponteiro aponta é inspecionada. Se seu bit R for igual a 0, a página é removida, a nova página é inserida no relógio em seu lugar, e o ponteiro avança uma posição. Se R for 1, R é resetado e o ponteiro passa apontar para a próxima página.
- Esse processo é repetido até encontrar uma página com R = 0, Devido a essa natureza, esse algoritmo recebe o nome de **algoritmo do relógio**.

1.5.5 Algoritmo LRU (Least Recently Used)

- Páginas que foram muito usadas nas últimas instruções provavelmente serão usadas novamente em breve **(princípio da localidade)**.
- Da mesma forma, páginas que não foram usadas há um bom tempo, provavelmente não serão usadas em breve.
- Essa idéia sugere um algoritmo: quando uma page fault ocorre, joga-se fora a página que tenha estado inativa pelo maior tempo. Essa estratégia é chamada de paginação **LRU (Least Recently Used)**.
- Apesar de LRU ser implementável, não é um altoritmo barato. Para implementar um LRU completo, é necessário manter uma lista encadeada de todas as páginas em memória, com a página mais recentemente utilizada na frente e a página menos recentemente utilizada no final. A dificuldade está no fato de que esta lista deve ser atualizada toda vez que a memória é referenciada.
- Encontrar uma página na lista, deletá-la, e então movê-la para frente é uma operação que consome muito tempo, mesmo em hardware.

1.5.6 Simulando LRU em Software

- **NFU (Not Frequently Used)**
  - A cada pagina é associado um contador (inicialmente zerado).
  - A cada interrupção de clock, o SO escaneia todas as páginas em memória. Para cada página, o bit R (que pode ser 0 ou 1) é adicionado ao contador.
  - Os contadores mantém um certo controle sobre o quão frequentemente cada página é referenciada.
  - Quando um page fault acontece, a página com o menor contador é escolhida para ser substituída.
  - O maior problema do NFU é que esses contadores nunca são zerados, e por isso, caso uma página que tenha sido bastante utilizada pare de ser utilizada, o valor do seu contador ainda poderá ser o maior de páginas que começaram a ser utilizadas agora.
- **Aging**
  - Uma pequena modificação no NFU permite simular LRU de forma eficiente.
  - Primeiro, todos os contadores são deslocados à direita 1 bit antes do bit R ser somado. Em seguida, o bit R é somado ao bit mais à esquerda, ao invés do mais à direita.
  - Quando um page fault ocorre, a página com o menor contador é removida. Caso uma página não tenha sido referenciada por 4 ciclos de clock, ela terá 4 zeros à esquerda, e por isso, um valor de contador menor do que uma página que não foi referenciada por 3 ciclos de clock.

1.5.7 Algoritmo Working Set

- Na forma mais pura de paginação, processos são iniciados sem nenhuma página em memória.
- Asim que a CPU tenta buscar a primeira instrução, ela encontra um page fault, e precisa fazer o SO buscar a página contendo a primeira instrução.
- Outras páginas contendo variáveis globais e a pilha normalmente vem em seguida.
- Depois de um tempo, o processo tem a maioria das páginas que precisa e se estabiliza, rodando com um número relativamente pequeno de page faults.
- Essa estratégia é chamada de **demand paging**, porque páginas são carrecadas apenas sob demanda, e não previamente.
- É fácil escrever um programa de teste que leia todas as páginas de um grande espaço de endereçamento, causando tantos page faults que não haja mais memória para segurar todos eles. Felizmente, processos não funcionam assim.
- Processos exibem **localidade de referência**, que significa que durante qualquer etapa da execução, o processo referencia apenas uma fração de todas as páginas.
- O conjunto de páginas que um processo atualmente está usando é seu **working set**. Se todo o working set está em memória, o processo irá rodar sem causar muitos page faults até seguir para outra fase de execução.
- Se a memória for pequena demais para armazenar todo o working set, o processo irá causar muitos page faults e rodar devagar, já que executar uma instrução exige alguns nanosegundos e ler uma página em disco tipicamente leva 10 msec.
- Um programa que causa muitos page faults frequentes é dito **thrashing**.
- Em sistemas multiprogramados, processos frequentemente são movidos para o disco (i.e. todas as páginas removidas da memória) para entregar a CPU para outros.
- Se, ao retomar a execução desse processo, for necessário causar page faults até que todo o working set esteja novamente disponível em memória, isso causará degradação no desempenho.
- Por isso, muitos sistemas de paginação tentam manter controle sobre o working set de cada processo, e ter certeza de que estará em memória antes de deixar o processo rodar. Isso é chamado de **modelo do working set**, e é concebido para diminuir a taxa de page faults. Carregar as páginas *antes* de deixar o processo rodar também é chamado de prepaging.
- Uma forma comum de se implementar uma aproximação de working set é usar o tempo de execução do processo. Por exemplo, podemos definir o working set de um processo como sendo o conjunto de páginas usadas durante os ultimos 100 ms de tempo de execução.
- O único tempo de execução que nos intereça é o tempo de execução do processo, e não o tempo de CPU total durante a execução do processo. Isso é, se um processo começa a

executar no instante T, e teve 40ms de tempo de CPU ao final de T + 100ms, para fins de escolha de working set, seu tempo é de 40ms.

- Esse tempo de CPU que foi realmente usado pelo processo desde que ele iniciou é chamado de **tempo virtual atual**. Com essa aproximação, o working set de um processo são as páginas que foram referenciadas durante os últimos τ segundos de tempo virtual.
- Um algoritmo de substituição de páginas baseado em working sets usaria a idéia básica de encontrar uma página que não esteja no working set e removê-la. Como apenas páginas presentes em memória são candidatas a serem removidas, págians que não estejam em memória são ignoradas por esse algoritmo. Cada entrada contem pelo menos dois itens de informação: o tempo aproximado que a página foi usada pela última vez e o bit R. Outros campos como o número da moldura de página, os bits de proteçao e o bit M são desnecessários para esse algoritmo.
- O algoritmo funciona da seguinte forma:
  - Assume-se que o hardware seta os bits R e M.
  - Assume-se que uma interrupção periódica de clock faz com que software limpe o bit R a cada ciclo de relógio.
  - A cada page fault, a tabela de páginas é escaneada para procurar uma página apropriada para ser removida.
  - O bit R de cada entrada é examinado. Se for 1, o tempo virtual atual é escrito no campo "tempo do ultimo uso"na tabela de páginas, indicando que a página estava sendo usada quando a falta ocorreu.
  - Como a página foi referenciada durante esse ciclo de clock, ela faz parte do working set e não é um candidato a ser removido.
  - Se R for 0, a página não foi referenciada durante o ciclo de clock atual e pode ser um candidato a ser removido. Para decidir se é removida ou não, sua idade (o tempo virtual atual menos o tempo de último uso) é computada e comparada com τ.
  - Se a idade for maior que τ, a página não faz mais parte do working set, e é substituída. As outras páginas são atualizadas em seguida. Porém, se R for 0 mas a idade for menor ou igual a τ, a página ainda faz parte do working set e é temporariamente mantida. Porém, a página que tiver a maior idade é anotada. Se a tabela inteira for percorrida sem encontrar um candidato para remover, isso significa que todas as páginas são parte do working set. Nesse caso, se uma ou mais páginas com R = 0 foram encontradas, a página que tiver a maior idade é removida.
  - No pior cenário, todas as páginas tem R = 1, e nesse caso a escolha é aleatória, preferindo uma página limpa, se existir.

1.5.8 Algoritmo WSClock

- O **WSClock** é um algoritmo baseado no algoritmo do relógio, mas que usa as informações de working set. Devido à sua simplicidade de implementação e performance, é altamente usado na prática.

- A estrutura de dados usada é uma lista cirular de molduras de página, assim como no algoritmo do relógio. Inicialmente, essa lista está vazia. Quando a primeira página é carregada, ela é adicionada a lista, e assim sucessivamente, formando um círculo.
- Cada entrada possui os campos *tempo de último uso*, bit R e bit M.
- Assim como no algoritmo do relógio, a cada page fault a página apontada pelo ponteiro do relógio é examinada primeiro. Se o bit R for 1, a pagina foi usada durante o ultimo ciclo de clock e por isso não é uma boa candidata a ser removida. O bit R é então resetado para 0 e o ponteiro avança para a próxima página, repetindo o procedimento.
- Se a página tem R=0, a idade dela for maior que τ e a página está limpa, então ela não faz parte do working set e possui uma cópia válida em disco, sendo substituida imediatamente. Caso a página esteja suja, ela não pode ser removida imediatamente pois precisa ser copiada para o disco. Para evitar que o processo seja trocado, essa escrita em disco é agendada para mais tarde, mas o ponteiro simplesmente avança e o algoritmo continua procurando uma página velha e limpa que possa ser usada imediatamente.
- É possível que todas as páginas sejam agendadas para escrita em disco em uma única "passada"no relógio. Para reduzir utilização de disco, um limite pode ser estabelecido, permitindo um máximo de n páginas agendadas para escrita. Após esse limite ser atingido, mais nenhuma escrita seria permitida.
- Se o ponteiro der uma volta completa e voltar ao seu ponto inicial, duas situações podem ter ocorrido:
  1. Pelo menos uma escrita foi agendada. Nesse caso, o ponteiro simplesmetne continua rodando, pois se pelo menos uma escrita foi agendada, ela eventualmente será realizada, e assim uma página limpa estará disponivel para ser substituída.
  2. Nenhuma escrita foi agendada. Nesse caso, todas as páginas pertencem ao working set. Então, procura-se qualquer página limpa para subsituir. Caso nenhuma exista, a página atual é substituída.

1.6 Segmentação

- Na paginação, memória virtual é unidimensional, e o endereço virtual vai de 0 até algum endereço máximo.
- Ter 2 ou mais espaços de endereçamento separados pode ser vantajoso, pois programas possuem regiões de código, dados, pilha, etc, que podem ser realocados mais facilmente na memória.
- Na segmentação, o espaço de endereçamento do processo é dividido em segmentos, que são blocos de endereços contíguos completamente independentes de tamanho variável (ex.: segmento de dados, código, pilha).
- O SO mantém uma tabela de segmentos para cada processo, a qual relaciona a memória virtual do processo com a memória física do computador.

- Essa tabela contém o número do segmento, endereço físico inicial do segmento e o tamanho do segmento.
- Endereços são divididos em 2 partes: número do segmento e deslocamento dentro do segmento.
- O endereço físico é obtido somando base + deslocamento.
- A tabela de segmentos é armazenada na memória RAM do computador, com registradores guardando os endereços base e limite da tabela de segmentos.
- Quando ocorrem trocas de contexto, é necessário recuperar valores dos registradores armazenados no descritor de processo.
- Vantagens em relação à paginação:
  - Tamanho variável: fácil de gerenciar estruturas de dados que crescem ou diminuem de tamanho
  - Ligação de métodos/funções compilados separadamente é muito simplificada, pois o ponto de entrada é sempre o mesmo: endereço 0.
  - Lovre de fragmentação interna
  - Fácil de compartilhar memória.
- Desvantagens:
  - Fragmentação externa: espaço desperdiçado entre segmentos
  - Inconveniente manter segmentos muito grandes em memória

1.6.1 Segmentação paginada

- Oferece vantagens das técnicas de paginação e segmentação, resolvendo o problema de manter em memória segmentos muito grandes.
- Utiliza-se uma tabela de segmentos, que aponta para tabelas de páginas dos diferentes segmentos.
- O endereço virtual consiste de um número do segmento, um número de página virtual e um deslocamento.
- O número do segmento endereça a tabela de segmentos. O valor encontrado na tabela de segmentos aponta para a tabela de páginas apropriada, que é endereçada pelo número de página virtual do endereço virtual. Nessa tabela de páginas se encontra o número da moldura de página, que é concatenado com o deslocamento para gerar o endereço físico.