Unidade III

5 GERENCIAMENTO DE PROCESSOS

5.1 Introdução a processos

Os sistemas computacionais atuais são capazes de desenvolver uma grande variedade de tarefas simultaneamente. Muitas vezes isso passa despercebido para nós e só nos lembramos desta extraordinária capacidade quando ela falha, ou quando o sistema já está sobrecarregado e notamos a degradação de desempenho em alguma das tarefas que lhe demos e da qual não estamos tendo o nível de serviço esperado.

Se pegássemos um servidor de arquivos como exemplo e pudéssemos ver o que está acontecendo dentro do cérebro dele no momento de pico de uso do sistema, notaríamos que dezenas, em muitos casos centenas, de usuários estão fazendo milhares de requisições simultaneamente, que o antivírus está varrendo tudo que entra e sai, e, por outro lado, temos o *software* de gerenciamento de redes monitorando e solicitando dados de desempenho do servidor e, por sua vez, os diversos discos trabalhando em RAID necessitam processar as funcionalidades desta tecnologia. Certamente, é necessário que os processos estejam em plena ação para orquestrar toda essa estrutura.

Em todos os sistemas com suposto conceito de paralelismo, temos a *CPU* trabalhando por algumas dezenas ou centenas de milissegundos numa única aplicação e subsequentemente na próxima até o ciclo se completar. Esta sequência continuará até termos todos os processos concluídos, porém não podemos esquecer que outros processos podem estar entrando na fila a todo o momento. Devido à rapidez dos ciclos, esse cenário de suposto paralelismo irá gerar para a percepção humana a sensação de que o ambiente está processando as requisições, atendendo às diversas aplicações de forma simultânea.

Somente nos casos de sistemas com múltiplos processadores é que teremos de fato múltiplos programas sendo atendidos no mesmo instante. Controlar múltiplas atividades em paralelo é algo que vem sendo desenvolvido e aprimorado com base num modelo conceitual de processos sequenciais que facilita o paralelismo que estudaremos neste capítulo.

5.1.1 Processo

Veremos neste capítulo que os *softwares* de computador são organizados em processos sequenciais.

> Um processo é um programa em execução, acompanhado dos valores correntes do contador de programa, dos registradores e variáveis.[1]

[1] Disponível em: <http://www.ufpi.br/subsiteFiles/eml/arquivos/files/SO/SO_Livro.pdf>. Acesso em: 8 jun. 2011.

Conforme demonstrado na Figura 31, uma *CPU* pode executar um processo por vez. Na Figura 32, temos a ilustração de um sistema com dois *cores*, que equivale a termos duas *CPUs*, porém ainda temos a mesma afirmativa – todas as *CPUs* podem executar um processo por vez.

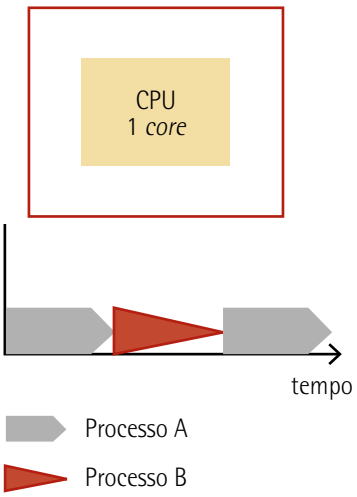


Figura 31 – *CPU* 1 *core* e 2 processos

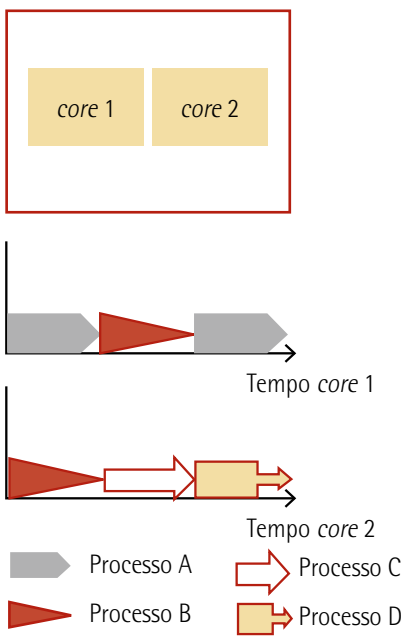


Figura 32 – *CPU* 2 *cores* e 4 processos

Vale destacar que um processo e um programa possuem conceitos distintos, sendo que o processo constitui uma atividade, possuindo programa, entrada, saída e um estado. Também devemos salientar que processos podem conter mais de uma tarefa, conceituando então que tarefa e processo são distintos.

No núcleo dos sistemas operacionais, temos o PCBs (Process Control Blocks) que armazena as informações referentes aos processos ativos no ambiente. Cada processo possui um identificador único no sistema, o PID – Process IDentifier.

5.1.2 Criação de processos

Processos são criados e destruídos constantemente nos sistemas. Essas operações disponibilizam aplicações por meio de chamadas de sistema que diferem entre sistemas operacionais.

Para os sistemas de propósitos gerais, é necessário algum mecanismo para criar e terminar processos durante a operação quando for necessário.

Teremos nos sistemas quatro eventos que fazem que processos sejam criados: no início do sistema, um processo em execução procedendo a uma chamada de sistema de criação de um processo, requisição do usuário para criar um novo processo e *bacth job*[2] sendo iniciada.

Ao iniciar o sistema operacional, tipicamente vários processos são criados. Entre esses processos, temos os que estão em primeiro plano e interagindo com o usuário e outros que estão em segundo plano, portanto não estão diretamente interagindo com o usuário. Para exemplificar um processo em segundo plano, podemos pegar o caso de um servidor de FTP (*File Transfer Protocol*) que fica inativo durante boa parte do tempo, sendo ativado somente quando um cliente FTP solicita a abertura de uma nova conexão – usamos o termo *daemons* para descrever um processo que fica em segundo plano com finalidade de lidar com alguma atividade como a descrita.

Processos que estão em execução podem fazer chamadas de sistema (*system calls*) para criar um ou mais novos processos. Criar novos processos é indicado quando a tarefa a ser executada puder ser facilmente dividida em vários processos relacionados, interagindo, entretanto, de maneira independente.

Os usuários podem iniciar um novo processo começando um programa no ambiente *GUI* ou no ambiente *Shell*.

No caso de sistemas em lote, tipicamente encontrados em computadores de grande porte, o usuário, administrador ou até mesmo um alinhamento prévio, pode submeter tarefa em lote para o sistema. O sistema operacional criará um novo processo e o executará quando tiver recurso disponível e/ou redefinindo prioridades e executando o processo no momento determinado.

Se usarmos como exemplo o ambiente Unix, teremos a chamada de sistema *fork* para criar um processo. Essa chamada cria uma réplica do processo solicitante conforme descrito na Figura 33.

[2] *Bacth job* = tarefa em lote.

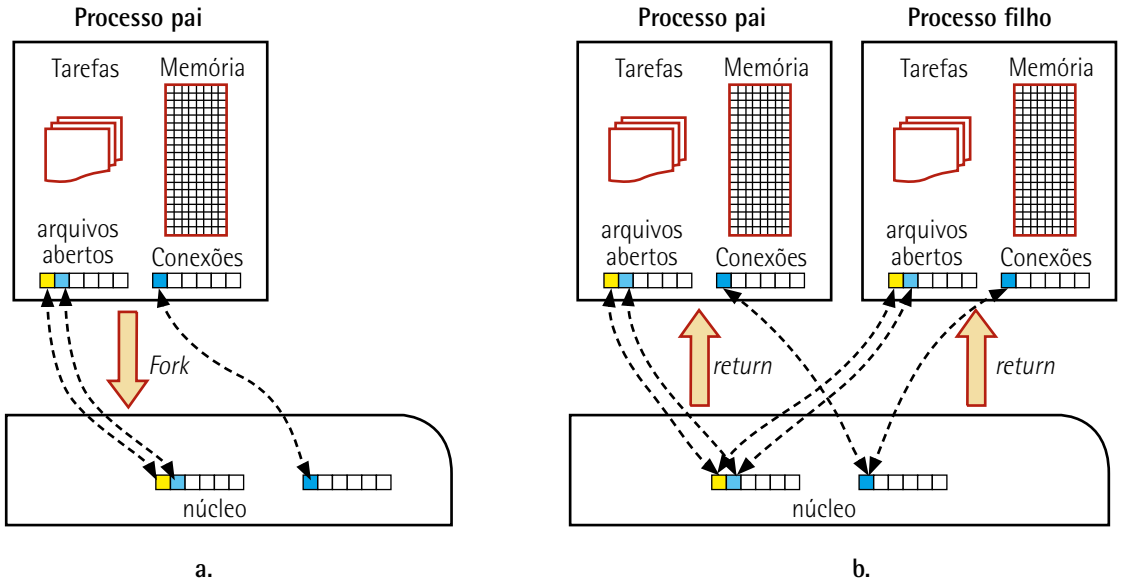


Figura 33 – A chamada de sistema *fork*: (33a. antes e 33b. depois).

Posteriormente, o processo filho executará subsequentemente "*execve*" ou uma chamada de sistema similar para mudar sua imagem de memória e executar um novo programa.

No ambiente Windows, uma única chamada denominada *CreateProcess* de função do Win32 trata o processo de criação e carga do programa correto no novo processo. O processo Win32 possui dezenas funções para gerenciar e sincronizar processos e tópicos relacionados.

Tanto no Unix quanto no Windows quando um novo processo filho é criado, o processo pai e filho possuirão seus próprios espaços de endereçamento de memória, permitindo assim que se o processo pai ou filho alterar uma palavra em seu espaço de endereçamento, a mudança não impacte o outro processo.

5.1.3 Término de processos

Após o término, o processo é finalizado com base nas quatro condições típicas: normal, por erro, erro fatal e cancelado por terceiros – sendo as duas primeiras voluntárias e as duas últimas involuntárias.

Processos terminados de forma involuntária não são comuns num sistema em perfeito funcionamento. A seguir, verificaremos as quatro condições e notaremos por qual motivo essa afirmação é um fato.

O primeiro caso, que é a condição normal de se encerrar um processo, é verificado pela chamada *exit* no Unix ou *ExitProcess* no Windows. Nestes casos, o processo termina após finalizar as tarefas que estavam previstas, mesmo que seja um usuário finalizando um programa, fechando a janela no ambiente *GUI* ou pela opção relativa no ambiente *Shell*.

Num ambiente Unix, a chamada de sistema *exit* serve para informar ao núcleo do sistema operacional que o processo em questão não é mais necessário e pode ser eliminado, liberando todos os

recursos a ele empregados. Processos podem solicitar ao núcleo o encerramento de outros processos, mas essa operação só é aplicável a processos do mesmo usuário ou se o processo solicitante pertencer ao administrador do sistema.

Os processos que interagem com outros não podem ser concluídos quando algum parâmetro errado é fornecido. Para exemplificar, vamos considerar o caso de um usuário tentando colocar o nome duplicado entre dois arquivos no sistema, então uma caixa de diálogo emerge e pergunta ao usuário se ele quer tentar novamente; desta forma teremos por consequência a segunda condição que é a saída por erro.

Erro fatal é um erro causado pelo processo e normalmente por um erro de programa. Como exemplo podemos ter a execução de uma instrução ilegal, a referência à memória inexistente ou a divisão por zero; em todos estes casos, teremos como resultado um erro fatal.

O cancelamento por outro processo ocorre quando um processo x executa uma chamada de sistema determinando que o sistema operacional cancele outro(s) processo(s) n. Tanto no Unix/Linux a chamada é o *kill* e no ambiente Windows a função Win31 correspondente é a *TerminateProcess*.

5.2 Comunicações entre processos

A comunicação entre processos é algo frequente nos sistemas atuais, havendo a necessidade de obtermos uma comunicação estruturada e sem interrupções acontecendo entre eles.

Em linhas gerais, temos três tópicos importantes na comunicação entre processos: como um processo passa a informação para outro; como garantir que múltiplos processos não entrem em conflito; e o terceiro é pertinente, como haverá uma sequência adequada quando existirem dependências.

5.2.1 Condição de corrida

Processos que trabalham juntos podem compartilhar algum armazenamento comum e serem capazes de ler e escrever. O armazenamento compartilhado pode estar na memória principal ou em um arquivo compartilhado.

Para entendermos o processo de condição de corrida, vamos considerar como exemplo um *spool* de impressão.

Para imprimir um arquivo, um processo entra com o nome do arquivo numa posição da fila em um diretório de *spool*. Em paralelo e de forma constante, o *deamon* de impressão verifica na fila se há algum arquivo para imprimir. Se houver algum arquivo para imprimir, ele será impresso e, em seguida, seu nome será removido da fila.

Ainda pensando no cenário para exemplificar nossa condição de corrida, imagine que há duas variáveis compartilhadas, sendo uma de saída com o nome *out* que apontará para o próximo arquivo a ser impresso, e uma de entrada como o nome de *in* que apontará para a próxima posição livre no diretório de impressão.

Agora, imagine que temos dois processos *X* e *Y* que decidem quase que simultaneamente colocar, cada um deles, um arquivo, sendo *X* = *ArquivoX* e *Y* = *ArquivoY*, na fila de impressão. Então, o processo X lê a variável *in* e nota que a posição 1 está disponível, o processo *X* armazena a posição 1 na sua variável *vaga_impressao_disponivel.* Porém, neste instante, a *CPU* transaciona para o processo *Y*, entendendo que *X* já ocupou o tempo necessário durante este ciclo de processamento. No momento em que *Y* lê a variável *in,* nota que a posição 1 está disponível, então *Y* armazena em sua variável *vaga_impressao_disponivel* a posição 1. Podemos notar que ambos os processos passam a conter a posição 1 como variável indicando a vaga 1 da fila de impressão.

O processo *Y* continua sua execução armazenando o nome *ArquivoY* na vaga 1, então atualiza a variável *in* para conter o valor 2. Depois desta tarefa, o processo *Y* passa a executar outras tarefas subsequentes. De forma fortuita, o processo *X* volta a executar suas tarefas da posição em que havia parado, após verificar sua variável *vaga_impressao_disponivel* e encontrar o valor 1, então escreve o *ArquivoX* na vaga 1. Após o processo *X* executar a tarefa de gravação do arquivo, teremos a posição 1 da fila de impressão ocupada pelo *ArquivoX* e o *ArquivoY* concomitantemente apagado. Como *X* atualiza a variável *in* com o valor 2, o diretório de *spool* está internamente consistente, não havendo nenhuma suspeita, portanto o *deamon* não notará nada anormal ou inconsistente, porém o processo *Y* jamais obterá qualquer saída.

O usuário que mandou imprimir o ArquivoY certamente imaginará que houve algum boicote após esperar por muito tempo e nada sair da impressora. Portanto, cenários como este em que temos dois ou mais processos que estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende de quem executa – e quando executa –, são chamados de condições de corrida (*race conditions*). É extremamente exaustiva a atividade de análise, depuração e resolução de códigos de programa que apresentam condições de corrida.

Saiba mais

Diagrama de Processos:

<http://www.las.ic.unicamp.br/edmar/PUC/2006/SO/SO-Aula2.pdf>

5.2.2 Exclusão mútua e região crítica

Exclusão mútua (*mutual exclusion*) é o modo de assegurar que processos sejam impedidos de usar uma variável ou um arquivo compartilhado que já estiver em uso por outro processo. Poderíamos ter evitado a condição de corrida apresentada anteriormente se tivéssemos aplicado a exclusão mútua ou por modo abstração evitando que os programas usassem a região crítica simultaneamente.

Entendemos por região crítica (*critical region*) ou seção crítica (*critical section*) a parte dos programas em que há acesso à memória ou arquivo compartilhado. Ainda que essa solução impeça as condições de disputa, isso não é suficiente para que processos paralelos colaborem de forma correta e eficiente usando dados compartilhados.

São necessárias quatro condições elementares para chegarmos a uma boa solução:

- Dois ou mais processos nunca podem estar simultaneamente em suas regiões críticas.
- Nada pode ser definitivamente afirmado no que tange à velocidade ou ao número de *cores*.
- Nenhum processo executado fora de sua região crítica pode bloquear outros processos.
- Nenhum processo deve esperar infinitamente para estar em sua região crítica.

Portanto, em um modelo abstrato, as características necessárias para satisfazermos a condição desejada são demonstradas na Figura 34, em que o processo *X* entra em sua região crítica no tempo *T1*. Subsequentemente, no tempo *T2*, o processo *Y* tenta entrar em sua região crítica, entretanto falhará devido ao fato de outro processo já estar ocupando esta condição exclusiva. Então, *X* fica temporariamente aguardando até que o processo *Y* deixe sua região crítica.

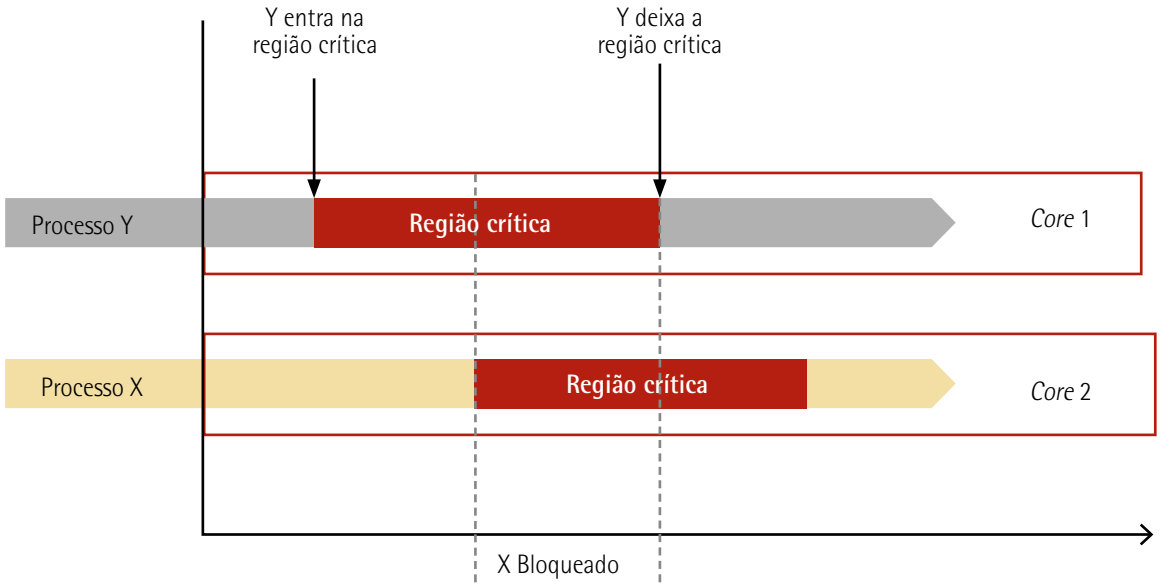


Figura 34 – Região crítica

5.2.3 Exclusão mútua com espera ociosa

Quando estamos usando os sistemas mais antigos que possuíam somente uma *CPU* com um único *core*, a forma mais trivial e segura para evitarmos que mais de um processo entre na região crítica é aplicada com a desativação das interrupções assim que o primeiro processo entrar na região crítica e consecutivamente reabilitá-las assim que sair desta região. Portanto, quando se desativa a interrupção, a *CPU* não poderá chavear para outro processo, com isso não tem como ocorrer a condição de corrida apresentada anteriormente.

Observação

Entretanto, caso tenhamos um problema num processo que desative as interrupções e nunca mais as reative, certamente, teremos um problema maior para lidar do que a situação decorrida, ou seja, teremos o sistema literalmente parado.

Podemos concluir que a desativação das interrupções é uma técnica coerente para o próprio sistema operacional, porém com alto nível de risco para os processos dos usuários que necessitem de exclusão mútua.

Com as novas arquiteturas de *chips* com múltiplos processadores, esta técnica de desabilitar interrupções torna-se inadequada, tendo em vista que, por um lado, se desativarmos a interrupção de uma única *CPU*, teremos outras *CPUs* ou *cores* com a possibilidade de terem processos entrando em regiões críticas e, por outro lado, se desativarmos a interrupção de todas as *CPUs* e/ou *cores*, estaremos impactando na *performance* da máquina, pois certamente comprometeremos as tarefas dos *chips* cujas interrupções desativamos por "precaução".

Existem outras técnicas para evitarmos que múltiplos processos entrem simultaneamente em suas regiões críticas. Entre elas, temos a opção por *software*. Essa opção seria baseada numa variável compartilhada que se chamaria *lock* (trava), em que teríamos valores 0 ou 1, sendo 0 para sinalizar que não temos nenhum processo em região crítica, e 1 para sinalizar que temos algum processo prévio em região crítica. O mecanismo é simples, um processo checa se está em 1 ou 0, estando em 0 então significa que tem permissão para entrar em sua região crítica, porém, "supostamente", não antes de colocar a variável em 1, o que indicaria que outro processo já está em sua região crítica. Portanto, se um novo processo tentar entrar numa região crítica, antes irá checar se a variável está em 1, estando assim, este processo irá aguardar. Entretanto, esta técnica apresenta o mesmo problema descrito no exemplo da fila de impressão.

Outra opção seria o chaveamento obrigatório que por meio da variável *turn* controla a vez de quem entra na região crítica, verificando e compartilhando a memória compartilhada. O funcionamento dessa técnica se dá quando um segundo processo que também queira entrar em sua região crítica terá de esperar até que a variável *turn* seja chaveada de valor. Porém, para isso precisa ficar testando de forma repetitiva até a liberação. Esse processo é chamado de *busy waiting* (espera ociosa).

Observação

O *busy waiting* é um processo que despende muito tempo de *CPU*, mas somente em situações nas quais a espera é factualmente pequena. Vale destacar que uma trava que usa o *busy waiting* é chamada de *spin lock* (trava giratória).

Num nível mais físico (*hardware*), temos a instrução *Test and Set Lock* (TSL) que lê no registrador RX a palavra *lock* do conteúdo da memória e então armazena um valor diferente de zero no endereço de memória *lock*. Com o uso desse mecanismo, temos a solução para o caso dos múltiplos *cores*, pois a instrução TSL impede que outras *CPU*s acessem a memória enquanto ela não terminar a execução do seu processo em região crítica.

5.2.4 Semáforos

Como vimos anteriormente, quando um processo estivesse ativo e executando tarefas na região crítica, então outros deveriam ficar "dormindo" até o término dessa tarefa. O semáforo é o conceito proposto por E. W. Dijkstra para um tipo de variável inteira, objetivando contar o número de sinais de "acordar" salvos para o uso futuro. Um semáforo poderia conter o valor 0, indicando que nenhum sinal de acordar foi salvo, ou algum valor positivo, sinalizando que um ou mais sinais de acordar estivessem pendentes. Dijkstra propôs a existência de duas operações para os semáforos: P (*down*) e V (*up*), que são mnemônicos e fazem alusão a *sleep* e *wake up* (dormir e acordar).

A operação *down* (P) no semáforo verifica se o valor contido é maior que 0. Caso isso seja mesmo um fato, subtrairá um sinal de *up* (V) que estiver na variável e prosseguirá com as tarefas. Porém, se o valor contido na variável apresentar o valor igual a 0, então o processo será instruído para *sleep* (dormir), porém sem terminar o *down* "ainda" (esse é um ponto importante do semáforo que será debatido adiante).

Observação

Vale destacar, nesse momento, que verificar o valor sem alterá-lo e possivelmente ir dormir são tarefas que devem ser obrigatoriamente executadas nessa ordem, além de não poderem ser divididas.

Para evitarmos a condição de corrida, deve ser garantido que, ao iniciar uma operação de semáforo, nenhum outro processo poderá ter acesso a este até que a operação tenha terminado ou sido bloqueada.

Já na operação *up*, é incrementado no semáforo um valor. Porém, se um ou mais processos estivessem dormindo ao ser iniciada a operação *up* (lembrando que na operação *down* os processos não chegam a terminar o ciclo completo, ficando em estado dormente – como foi destacado acima quando usei o termo "ainda"), e estivessem impossibilitados de terminar a operação *down*, então um deles seria escolhido pelo sistema e atribuído a permissão para término do *down*. Apesar de o semáforo permanecer em 0 nessa etapa *up*, em um semáforo que já continha um ou mais processos dormentes, e esperar para terminar o *down*, teremos um processo a menos dormindo.

Lembrete

Durante o *up*, apesar de o semáforo terminar com o valor em 0, ele terá liberado um dos processos que estavam impossibilitados de terminar o ciclo *down*.

Não podemos esquecer que a operação de incrementar o semáforo e acordar um processo que estava previamente dormente também é indivisível, como o processo no caso explicado anteriormente no *down*. Vale destacar também que um processo nunca deverá ser bloqueado a partir de um *up*.

Saiba mais

<http://www.facom.ufu.br/~faina/BCC_Crs/INF09-1S2009/Prjt_SO1/semaphor.html>

5.2.5 Monitores

Hoare e Brinch Hansen propuseram uma unidade básica de sincronização de alto nível chamada de monitor. Podemos definir monitor como uma coleção de rotinas, variáveis e estruturas de dados, todos agrupados em um tipo especial de pacote.[3]

O monitor tem um papel fundamental para realizar a exclusão mútua pelo fato de que somente um processo pode estar ativo em um monitor num determinado tempo x. Tipicamente, quando um processo executa uma chamada a uma determinada rotina do monitor, algumas das primeiras instruções da rotina deverão verificar se existe outro processo ativo dentro do monitor. Caso confirme que outro processo encontra-se ativo dentro do monitor, então o processo que realizou a chamada ficará suspenso até que o processo que estava previamente ativo saia do monitor.

Observação

Um processo que executar uma chamada ao monitor poderá entrar somente se não houver nenhum outro previamente ativo.

O monitor é uma construção da linguagem de programação e os compiladores tratam suas chamadas e rotinas de modo diferente de outras chamadas de procedimento. Também é função do compilador implantar a exclusão mútua. Tendo em vista que para codificar um monitor, quem codifica

[3] Disponível em: <http://www.computronixbras.com/cursos/SOP/Windows2000/ApostilaSOSite.pdf>. Acesso em: 8 jun. 2011.

não necessariamente tem de conhecer como o compilador implanta a exclusão mútua, então basta ter em mente que convertendo todas as regiões críticas em rotinas do monitor, dois ou mais processos nunca poderão entrar em suas regiões críticas ao mesmo tempo.

Além da maneira simples pela qual o monitor consegue tratar as exclusões mútuas, ele também apresenta variáveis condicionais que possibilitam bloquear processos quando não puderem continuar.

As variáveis condicionais apresentam duas operações: *wait* e *signal*. O *wait* é usado para definir que uma rotina não pode prosseguir naquele instante. O *signal* foi abordado de duas formas distintas, sendo uma por Hoare, que propõe deixar o processo recém-acordado executar e suspender o outro. Já Brinch Hansen propôs que, se um *signal* é emitido sobre uma variável condicional pela qual os vários processos estejam esperando, somente um deles, que é determinado pelo escalonamento do sistema, será despertado. Além dessas formas há uma terceira que descreve a solução, deixando o emissor do sinal prosseguir sua execução e permitindo ao processo em espera começar a executar somente depois que o emissor do sinal tenha saído do monitor.

Podemos notar que no semáforo, usando o *sleep* e o *wake up*, havia a possibilidade de ocorrer falhas, porque pode haver uma situação, na qual um processo está tentando ir dormir e, em paralelo, outro tentando acordá-lo. Porém, quando usamos monitores, isso não tem como acontecer com a exclusão mútua que é automática. O *signal* não poderá acontecer até que o *wait* tenha terminado.

Linguagens como C, Pascal e outras diversas não possuem monitores. Entretanto, o monitor foi projetado para resolver o problema de exclusão mútua em CPU, acessando memória comum, porém, quando estamos usando um sistema distribuído formado por múltiplas CPUs, e cada uma com sua própria memória privada e conectada por uma rede, os monitores passam a não ter efeito.

Saiba mais

<http://www.deinf.ufma.br/~fssilva/graduacao/so/aulas/monitores.pdf>

5.2.6 Troca de mensagens

Semáforos e monitores não permitem troca de informações entre máquinas, que é primordial no mundo dos sistemas distribuídos. Para essa condição temos o *message passing* (troca de mensagens), que usa dois instrumentos: *send* e *receive* (envio e recebimento) colocados em rotinas de biblioteca.

Num ambiente de rede, um dos principais problemas é a perda de pacotes causados por algum motivador que estaremos nos abstraindo neste material de *Fundamentos de Sistemas Operacionais*. No caso da troca de mensagens entre máquinas, isso se dá por meio da rede e, como descrito anteriormente, essa mensagem pode ser extraviada ao longo do percurso. Portanto, uma troca de mensagem usa o

mecanismo similar ao usado no protocolo TCP/IP camada 4 Transportes TCP, onde a mensagem enviada requer um sinal de *acknowledge*[4], ou seja, se quem enviou não receber a confirmação, então uma nova mensagem será reenviada.

Vale destacar que se o emissor enviar a mensagem e o receptor receber, mas o problema acontecer no retorno da confirmação, então o emissor enviará uma nova mensagem. Entretanto, dessa vez, pelo fato de a mensagem ter um número sequencial de confirmação, o receptor irá identificar que é uma retransmissão e descartará a mensagem – o mesmo se aplica no protocolo TCP da camada de transporte do modelo TCP/IP.

5.2.7 Escalonamento

Quando temos uma única *CPU*, ou uma única *CPU* disponível entre as diversas existentes no sistema, e mais de um processo estiver competindo para ser executado, então caberá ao sistema operacional escolher qual dos processos será privilegiado e essa escolha chama-se algoritmo de escalonamento.

Em linhas gerais, o escalonamento é importante porque, dentre vários processos, é saudável que o sistema priorize aqueles que vão gerar mais impacto ao ambiente e seus usuários, caso não forem privilegiados durante a escolha de quem deve ser o próximo a ser processado.

Entre os processos, temos aqueles que passam a maior parte do tempo computando, *computer bound* (limitados pela *CPU*), enquanto outros passam a maior parte do tempo esperando por entrada e saída, *I/O bound* (limitados a entrada e saída).

Devemos escalonar os processos em quatro situações descritas na sequência abaixo:

1. Quando temos os processos pai e filho para serem executados. A definição de qual deve ser priorizado, em muitos casos, é essencial para o perfeito funcionamento das tarefas e resultado correto.

2. Quando temos um processo que terminou e já não está mais no sistema, há a necessidade da escolha de um novo processo e, portanto, o escalonamento, nessa situação, faz-se necessário.

3. Quando um processo é bloqueado por alguma razão, então outro processo deve ser selecionado para ser executado. Processos predecessores podem ser priorizados, pois, se forem executados os sucessores, pode haver dependências que irão gerar resultados inconsistentes.

4. Ao ocorrer uma interrupção de E/S, pode ser necessário uma decisão de escalonamento.

Os algoritmos de escalonamento podem trabalhar tipicamente de duas formas: não antecipado e antecipado. No primeiro caso, o processo "não antecipado" pode ficar executando pelo tempo

[4] *Acknowledge*: sinal que é enviado por um receptor para indicar que uma mensagem transmitida foi recebida e que ele está pronto para a próxima mensagem.

que for necessário, ou seja, por horas até que seja bloqueado ou até que libere a *CPU*. No segundo, o algoritmo de escalonamento antecipado escolhe um processo e o deixa em execução por tempo máximo fixado.

6 GERENCIAMENTO DE MEMÓRIA

6.1 Introdução a gerenciamento de memória

Até aproximadamente o primeiro quarto dos anos 2000, as memórias RAM (também conhecidas como memória principal) eram extremamente caras. Ficávamos completamente pasmos se comparássemos percentualmente o valor dos pentes de memórias em relação ao valor total do computador. Porém, já tivemos dias piores. Nos anos de 1960, mesmo os maiores computadores do mundo possuíam cada um algo que faria com que eles parecessem uma bica d'água e nossos computadores de mão, um rio Amazonas.

E como era possível termos programas tão eficientes com tão pouca memória? Na realidade, a resposta é dividida em duas partes. A primeira é que o programador daquela época também conhecia as entranhas do computador e programar era algo que exigia mais do que conhecimentos da linguagem de programação *versus* a necessidade de negócio que estava motivando aquele projeto. Por isso, o programador, tipicamente, tinha de "tirar leite de pedra". A outra parte da resposta é que não tínhamos as múltiplas interfaces entre aplicações em rede, nem ambientes de trabalho com tanta qualidade gráfica, nem mesmo a complexidade que os programas atuais possuem.

Com o passar do tempo, a hierarquia de memória, vista antes neste livro, contribuiu para atender a demanda exponencialmente crescente por memória. Podemos comparar os programas atuais com um camelo que, depois de atravessar o deserto por dias sem beber nenhum gole de qualquer coisa líquida e de uma exaustiva jornada, encontra um tanque d'água (a água seria a memória). Não é necessário concluir que, enquanto houver água disponível, o camelo estará consumindo – as aplicações atuais são cada vez mais cedentes por memória.

No sistema operacional, a parte parcialmente responsável por gerenciar a hierarquia de memória é o **gerenciador de memória**, que tem como tarefa conhecer todo espaço de memória, alocar para os processos que estão necessitando e liberar as partes que não estão mais em uso pelos processos.

Conforme demonstrado na Figura 35, temos algumas variações, sendo a primeira demonstrada na Figura 35a, a variação por uso total de memória RAM (*randon access memory* – memória de acesso randômico) onde o sistema operacional ficará na parte inicial e da memória e o programa do usuário na parte mais elevada. O segundo modelo é um *mix* de dois tipos de memória, sendo o programa do usuário na memória RAM e o sistema operacional contido numa memória ROM (*read only memory* – memória exclusiva de leitura) e como o nome já diz é somente para leitura, portanto neste modelo não há o risco do usuário ou algum problema na aplicação comprometer o sistema operacional – tipicamente este modelo é usado em sistemas portáteis. No terceiro caso apresentado na Figura 35c temos os *drivers* de dispositivos em ROM e programa do usuário e sistema operacional em RAM.

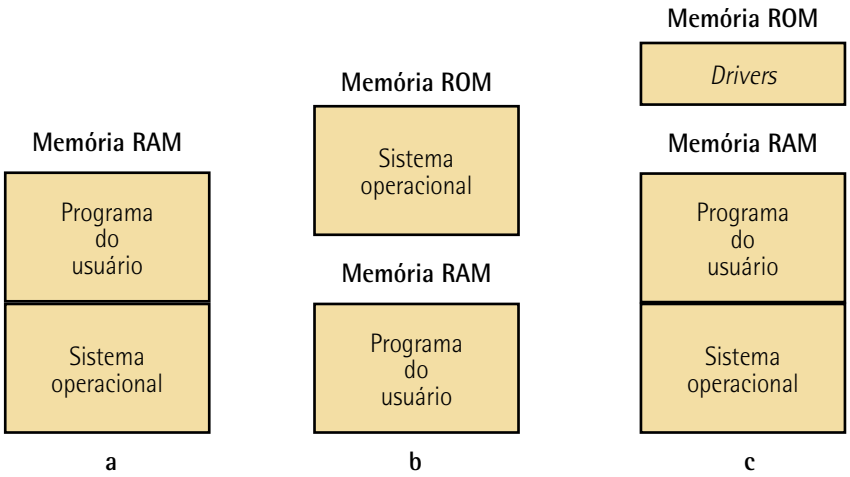


Figura 35 – Conceito de modos de organização de memória

Observação

Nos sistemas precursores não era possível mais que um programa ocupando a memória. Se isso ocorresse, causaria problemas aos dois programas, ao que estava na memória e àquele que tentasse fazer uso.

Com o avanço e a necessidade de múltiplas aplicações em funcionamento simultaneamente, a opção encontrada foi o uso do *swapping*, ou seja, troca de processos. Isso consiste em o sistema operacional pegar o conteúdo completo da memória e movê-lo para um arquivo na memória em disco rígido e, subsequentemente, liberar a memória para o próximo processo.

Entretanto, no *hardware* também houve avanço que não demandava somente a troca de processos executada por *software* (pelo sistema operacional). Esse processo consiste em dividir a memória principal em blocos de 2 KB com chave de proteção de 4 *bits* para cada bloco e mantidas em registradores especiais dentro da *CPU*.

Porém, nesse caso da divisão da memória em blocos, há "um problema quando se usa mais de um programa". Os dois programas referenciam a memória física absoluta, enquanto, na realidade, queríamos que cada programa referenciasse um conjunto de endereços. Portanto, a solução é a realocação estática ou, em outras palavras, esse mecanismo de rotação visa rotular blocos da memória com uma chave de proteção e comparar a chave do processo em execução com a de cada palavra da memória recuperada.

Saiba mais

<http://www.slideshare.net/audineisilva1/gerenciamento-de-memoria>

6.2 Abstração – espaços de endereçamento da memória

Expor a memória física aos processos pode trazer problemas, chegando até a ocasionar o travamento do sistema operacional. Entretanto, hoje, depois da história toda ter acontecido, sabemos que existe um método para tratar essa situação, caso contrário não seria possível estarmos com o nosso computador conectado à internet, acessando o editor de texto e/ou diversas combinações que fazem parte do nosso dia a dia. Para isso temos de entender o processo de abstração da memória.

Com a abstração da memória e a implantação do espaço de endereçamento, cria-se uma memória abstrata para abrigar os programas. Esta, por sua vez, possui um conjunto de endereços usado para que o processo realize endereçamento à memória. Individualmente, os processos possuem seu próprio espaço de endereçamento, diferente para cada processo.

6.2.1 Permuta de memória

Dezenas de processos são carregados somente para manter o sistema operacional típico funcionando, isto é, só para ligar a máquina e aparecer a tela do *desktop*, sem ao menos abrir um bloco de notas. Todos os processos dos sistemas atuais são verdadeiros consumidores esfomeados por memória. Um simples processo inerente do próprio Windows ou Linux pode consumir algumas dezenas de *megabytes* de memória. Para gerir todo esse alto consumo de memória, necessitamos de mais memória. Um paradoxo, não?

O *swapping*, previamente comentado, é um dos métodos mais triviais para gerir a sobrecarga de memória. Esse método vem sofrendo modificações ao longo dos anos. Ele faz a cópia completa do conteúdo da memória (que geralmente são processos ociosos) para um arquivo no disco rígido e libera a memória para outro processo ocupá-la. O outro método é o uso de memória virtual, permitindo que programas possam ser carregados na memória principal e executados na íntegra ou parcialmente.

Para entendermos o sistema de troca de processos, temos, na primeira etapa desse exemplo, em 36a, inicialmente, o sistema operacional ocupando a parte mais baixa da memória (permanecerá nessa posição) e, logo em seguida, temos o processo *X* ocupando uma parte da memória disponível. Depois, na Figura 36b, um novo processo *Y* é criado ou trazido do disco duro e posicionado na memória logo acima do processo *X*. Na Figura 36c, um novo processo *Z* é adicionado (notem que nesse momento não há mais espaço disponível na memória principal para novos processos). Na Figura 36d, o processo *X* fica ocioso, então é enviado para o disco rígido. Em 36e e 36f, temos outros processos sendo trocados e o ciclo vai sendo executado até que novos processos entrem e disputem o tempo de *CPU* e memória e/ou que processos terminados sejam eliminados do ciclo.

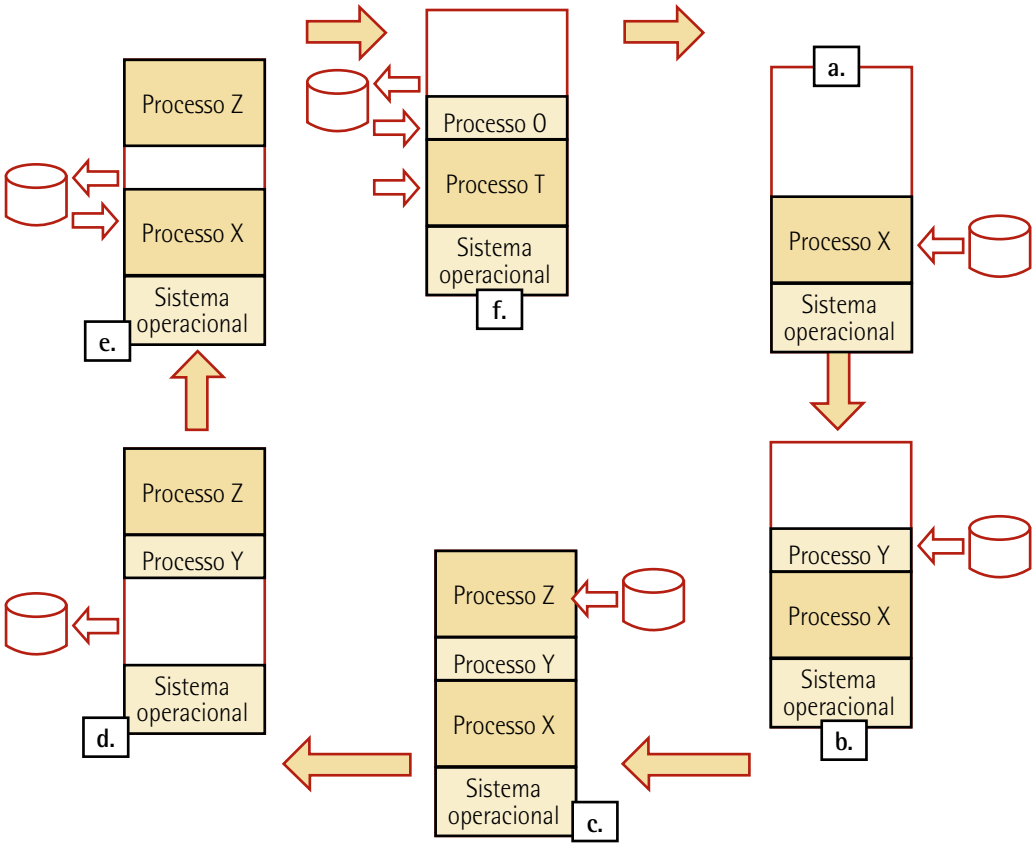


Figura 36 – Ciclo de troca dos processos entre memória principal e secundária

Observação

Se um processo maior que a área livre, ou um processo, mesmo que pequeno, porém, sem nenhuma área de memória, estiver disponível, então esse processo deverá ser transferido para disco e ficará por lá até que memória suficiente seja liberada.

6.3 Memória virtual

Diante da elevada demanda por memória, os programas eram criados em módulos denominados *overlays*[5]. Dessa forma, ao carregar um programa, o gerenciador de módulo era quem, na realidade, seria carregado e, em seguida, a sobreposição zero. Quando necessário, era carregado à próxima sobreposição desse programa ou de outros. Todas as sobreposições ficam gravadas em disco.

Em 1961, um método desenvolvido por John Fotheringham ficou conhecido como memória virtual que é um conceito extremamente importante no ambiente de ciência da computação, permitindo que programas usem mais RAM do que realmente está disponível fisicamente. Esse processo é possível porque o sistema operacional mantém rodando na memória principal somente as partes necessárias

[5] *Overlays*: módulos de sobreposição.

do programa e as outras, que não estão em uso, ficam no disco rígido. Quando é necessário que outra parte, que está no disco, seja carregada, então haverá o processo de *swapping*[6].

A memória virtual possui dois aspectos importantes: o primeiro é a quantidade de memória fisicamente instalada no equipamento, que chamamos de memória real. O outro tem muito mais capacidade que o primeiro e chamamos de espaço de memória virtual. No *hardware*, temos um componente de extrema importância que é a Unidade de Gerenciamento de Memória (MMU). O MMU[7] suporta o sistema operacional na execução do mapeamento dos endereços da memória física e endereços da memória virtual, permitindo, assim, a eficaz maestria de mover as partes dos programas da memória virtual para o disco ou vice-versa.

Observação

Analisando pela perspectiva do programa, temos cada um com seu próprio espaçamento de endereços adjacentes que chamamos de páginas.

6.3.1 Paginação

A técnica chamada paginação é usada na maioria dos sistemas de memória virtual. A memória virtual é dividida em unidades de espaçamento de endereços adjacentes chamadas de páginas. Estas correspondem a unidades das memórias chamadas de *frames*.

Observação

Enquanto o espaço de endereçamento virtual é dividido em unidades chamadas páginas (*pages*), temos as unidades correspondentes na memória física que são denominadas molduras de página (*frames*). Tanto as páginas quanto as molduras possuem o mesmo tamanho.

Se usarmos como exemplo um sistema que permite gerar endereços virtuais de 16 *bits*, – 2^{16} (de 0 a 64 K), entretanto, se esse ambiente possuir somente 32 KB de memória física, então isso significará que apesar de ser possível que programas de 64 KB sejam escritos, por outro lado não poderão ser carregados por completo na memória física. Sistemas reais possuem páginas de 512 *bytes* a 65.536 *bytes*. Neste exemplo, adotaremos páginas de 8.192 *bytes* (8 KB). Portanto, para atendermos a premissa que *frames* e páginas possuem o mesmo tamanho, teremos:

64 KB / 8 KB = 8 páginas virtuais

32 KB / 8 KB = 4 *frames*

[6] *Swapping*: sistema no qual um programa é movido para a memória secundária enquanto outro está sendo executado.

[7] MMU: *Memory Management Unit* – Unidade de Gerenciamento de Memória.

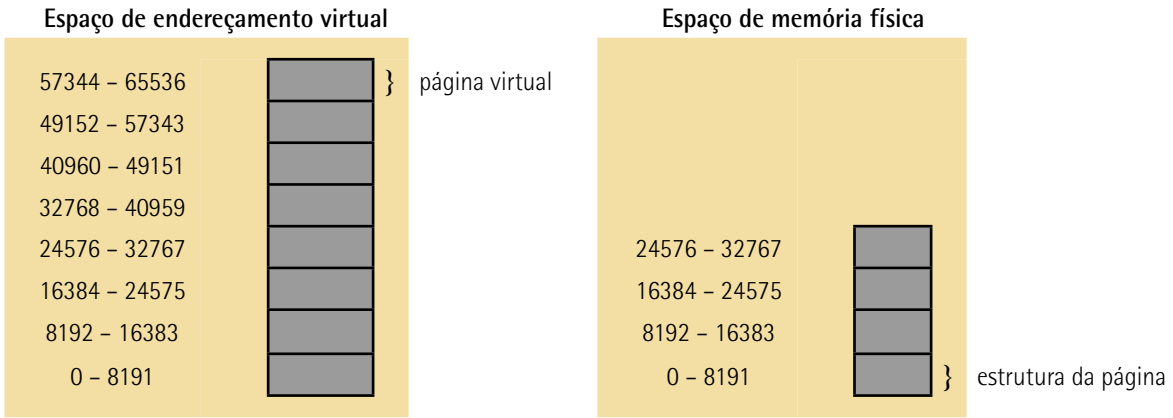


Figura 37 – Comparativo espaço de endereçamento virtual *versus* endereço de memória física

Nos sistemas em que não é utilizado o mecanismo de memória virtual, o endereço virtual é idêntico ao endereço físico. Portanto, para executar uma operação, o endereço virtual é colocado no barramento diretamente, ou seja, não é necessário que haja uma adequação do mundo virtual para o mundo físico. Entretanto, quando está presente o mecanismo de memória virtual, então o endereço virtual vai para o MMU que mapeia endereços virtuais em endereços físicos antes de colocá-lo no barramento. É mostrado na Figura 38a o fluxo de envio do endereço virtual, passando pelo MMU e envio do endereço físico para o barramento e na Figura 38b é demonstrado como é feito o mapeamento do endereço virtual para o endereço físico.

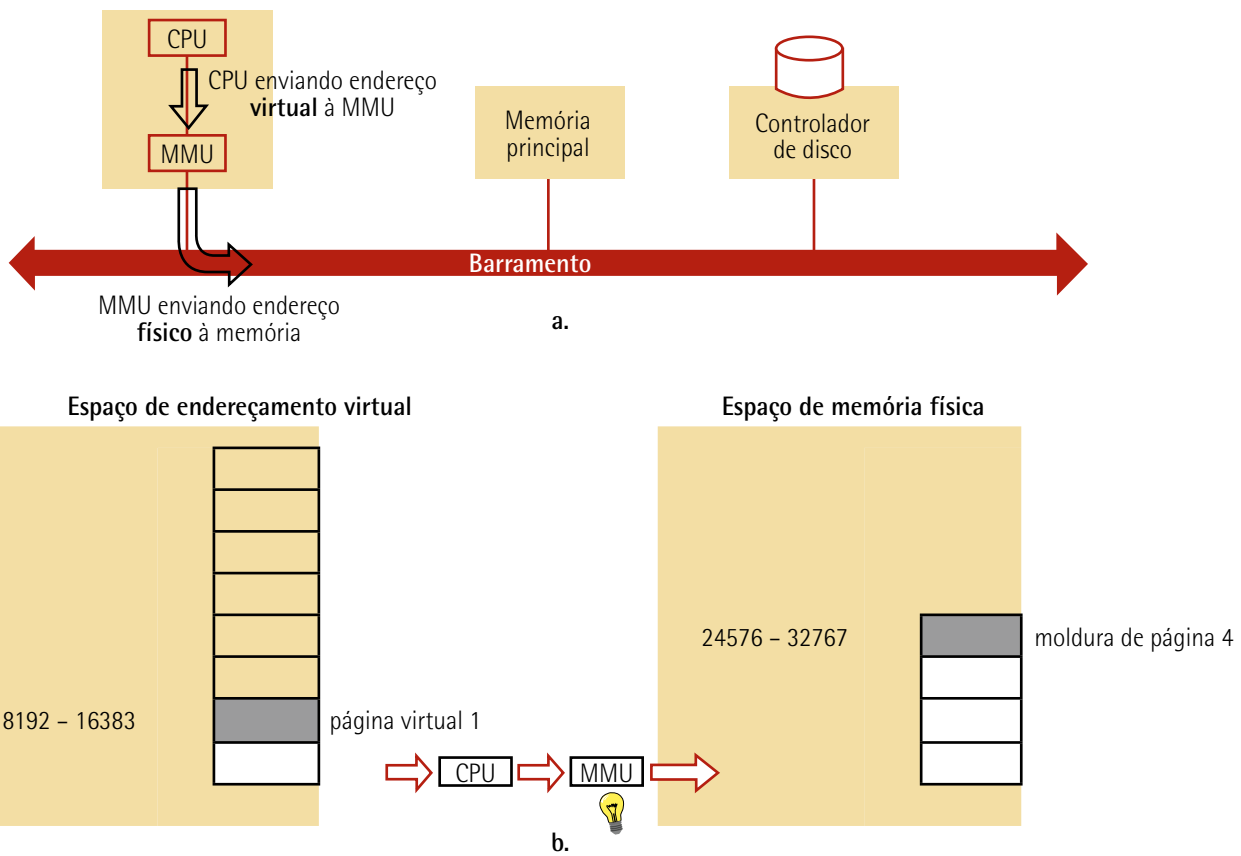


Figura 38 – Fluxo do endereço virtual *versus* endereço físico

Na Figura 38b, foi isolada a conversão da página 1 para a moldura de página 3, ou seja, quando um programa tenta acessar o endereço 8.192, então o endereço virtual "um" é enviado pela *CPU* para a MMU. Com base na tabela de páginas, a MMU detecta que a página virtual 1 corresponde à moldura de página 4 (de 24.576 a 32.767).

Desenvolvedores de sistemas operacionais encontraram outras formas para melhorar a paginação de memória virtual. Por exemplo, alguns sistemas podem restringir algumas páginas para serem utilizadas somente como leitura (*read-only*); na memória, as páginas exclusivas de leitura são usadas para armazenar código de programas, portanto não podem ser modificadas por vírus, por exemplo, e valores constantes que os programadores não estão autorizados a trocar.

Se qualquer tentativa de alteração em uma página restrita for executada por um programador, o resultado será uma situação de falha ao tentar alterar a página (*page fault*) ou erro de segmentação (*segmentation error*). Apesar de inicialmente esse aspecto proteger algumas páginas com atributo de somente leitura, com os ambientes em rede e diversos ataques sendo desferidos contra os sistemas, esses mecanismos passam a ser cada vez mais estudados pelos especialistas em segurança.

Saiba mais

<homepages.dcc.ufmg.br/~scampos/cursos/so/aulas/aula12_4.ps>

6.4 Segmentação

Além da paginação, a segmentação de memória é uma das formas mais simples para se obter a proteção da memória. Com o uso da segmentação são atendidos os seguintes requisitos:

1. Pode haver vários segmentos distintos.

2. Cada segmento pode ter um tamanho próprio.

3. Cada segmento é constituído de uma sequência linear de endereços.

4. O tamanho dos segmentos pode variar durante a execução.

5. O tamanho de cada segmento de pilha pode ser expandido sempre que algo é colocado sobre ela e diminuído sempre que algo é retirado dela.

6. Segmentos diferentes podem crescer ou diminuir independentemente e quando for necessário.

Quando o mecanismo de segmentação está presente e em uso, programas devem fornecer um endereço composto de duas partes:

1. Um número referindo-se ao segmento desejado.

2. Endereço dentro do segmento.

Além de um segmento conter o conjunto de permissões e o número de itens de dados em uma variável ou lista, um segmento também contém a informação indicando onde o segmento está localizado na memória, podendo conter, inclusive, uma identificação se o segmento está na memória principal ou secundária. Se o segmento requisitado não estiver na memória principal, uma exceção será enviada, então o sistema operacional irá trocar o segmento da memória secundária para a principal. A informação indicando onde o segmento está localizado na memória deve ser, primeiro, o endereço inicial do segmento na memória, ou deve ser o endereço da tabela de página para o segmento, se a segmentação é implantada com a paginação.

> A memória segmentada aparece para o programa como um grupo independente de espaços de endereço chamado de segmento. Código, dados e pilhas são tipicamente contidos em segmentos separados. Para endereçar um *byte* num segmento, um programa envia o endereço lógico.[8]

Este consiste em um número referenciado ao segmento desejado e um endereço dentro do segmento – um endereço lógico é também conhecido como ponteiro. A referência de segmento identifica o segmento para ser acessado e o endereço lógico identifica um *byte* no espaço de endereço do segmento. Programas rodando em um processador IA-32[9] podem endereçar até 16.383 segmentos com diferentes tamanhos e tipos, e cada segmento pode ser dimensionado na ordem de grandeza de 2^{32} *bytes* (4 GB). Internamente, todos os segmentos que são definidos para o sistema são mapeados dentro do espaço de endereço linear do processador. Para acessar a localização da memória, o processador traduz cada endereço lógico num endereço linear. Essa tradução é transparente para o programa que está usando a segmentação. A região primária para uso da memória segmentada é para aumentar o grau de confiança que se pode ter quanto ao desempenho de programas e sistemas. Por exemplo, alocando uma pilha de um programa em um segmento separado previne o crescimento da pilha lendo do espaço de código ou dado e sobreposição de instruções ou dados.

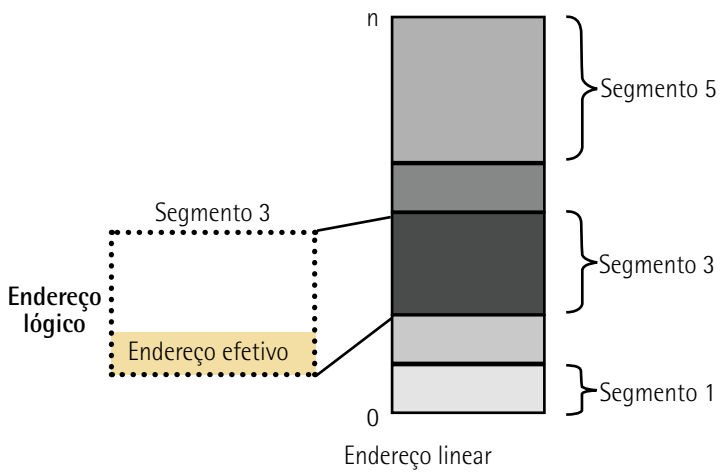


Figura 39 – Segmentação de memória

[8] Disponível em: <http://www.cpdee.ufmg.br/~fvasc/Disciplinas/Interfaces/Aulas/AULA_%B5P32bits.pdf>. Acesso em: 8 jun. 2011.

[9] IA-32: Intel Architecture, 32-bit, genericamente chamado de i386, x86-32 ou x86.

Na Figura 39, são demonstrados cinco segmentos de tamanhos e tipos diferenciados, em destaque ao lado esquerdo, tem a demonstração do segmento 3 que está parcialmente usado.

Resumo

Os processos são oferecidos pelos sistemas operacionais, ocupando cada qual o seu próprio espaço de endereçamento. Eles podem ser criados e terminados de maneira dinâmica, de forma a evitar que dois processos estejam em suas regiões críticas simultaneamente. Os semáforos, os monitores e as mensagens são as formas nas quais os processos comunicam-se entre si.

Estados do processo:

- Executando.
- Passível de ser executado.
- Bloqueado.

É possível que o processo troque de estado quando ele, ou um outro processo, executa uma das unidades básicas (semáforos, monitores ou mensagens).

Algoritmos de escalonamento são importantes para o ambiente e alguns sistemas fazem distinção entre mecanismo de escalonamento e política de escalonamento, permitindo aos usuários controle sobre o algoritmo de escalonamento.

Algoritmo de escalonamento é a escolha feita pelo sistema operacional de qual dos processos será privilegiado quando há uma única *CPU*, ou uma única *CPU* disponível entre as diversas existentes no sistema, e mais de um processo estiver competindo para ser executado.

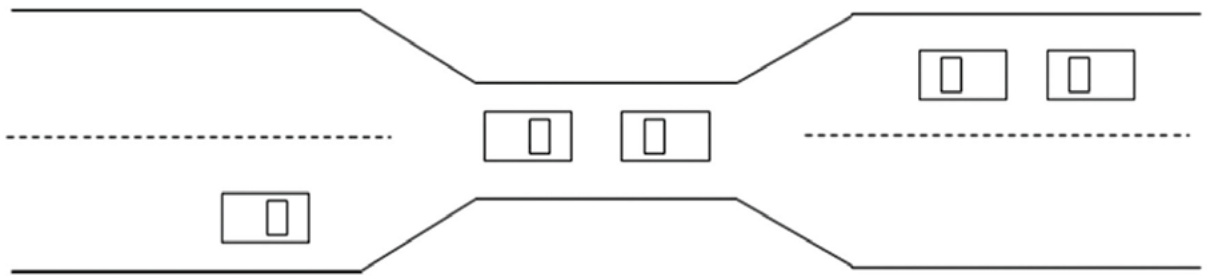
Nos sistemas mais triviais, quando um programa é carregado em memória, ele ficará ocupando a memória necessária até que sua finalização aconteça. Alguns sistemas permitem somente um processo por vez carregado na memória principal, enquanto outros suportam a multiprogramação.

Quando é necessário que o sistema operacional use mais memória principal (RAM), que realmente existe fisicamente na máquina, então é necessária a troca de processos entre a memória principal e o disco.

Os espaços de endereçamento de cada processo são divididos em blocos e são chamados de páginas. Na memória fica a moldura de página que recebe os blocos dos processos. O sistema de paginação ajuda no tratamento de estrutura de dados que alteram seus tamanhos durante a execução, simplificando a ligação, o compartilhamento e facilitando a proteção customizada para cada segmento. Segmentação e paginação são tipicamente combinadas para fornecer uma memória virtual bidimencional.

Exercícios

Questão 1. Considere o exemplo do cruzamento da ponte ilustrado na figura abaixo e descrito nos itens a seguir.



- Se ocorrer a disputa entre os carros na parte central da ponte, haverá um problema de sincronismo.
- Cada seção de uma ponte pode ser vista como um recurso.
- Se ocorrer um *deadlock*, ele pode ser resolvido se um dos carros recuar (*preemptar* recursos e reverter).
- Vários carros podem ter de recuar se um *deadlock* ocorrer.
- É possível haver *starvation*.

O problema acima pode representar duas situações envolvendo sincronismo, *deadlock* e *starvation*. O que poderá acontecer se ocorrer *deadlock*?

A) Os dois carros estão representando dois processos que aguardam, indefinidamente, por um evento que só poderá ser causado por um desses processos.

B) Os dois carros estão representando dois processos bloqueados indefinidamente. Um processo pode nunca ser removido da fila de semáforos em que está suspenso.

C) Se o processo Pi, representado por um dos carros, está executando em sua seção crítica, então nenhum outro processo poderá estar executando em suas próprias seções críticas.

D) O carro da direita irá subir na mureta, enquanto o segundo carro poderá passar. O segundo carro irá disparar um processo de agradecimento pela atitude, dando dois toques na sua buzina.

E) O carro da esquerda irá compartilhar sua seção crítica com o carro da direita sempre que ele necessitar.

Resposta correta: alternativa A.

Análise das alternativas

A) Alternativa correta.

Justificativa: durante um impasse entre dois processos, somente a intervenção de um deles pode desbloqueá-los.

B) Alternativa incorreta.

Justificativa: um semáforo funciona como uma variável que atribui 0 ou 1 aos processos que estão dormindo, não existe uma fila de semáforos.

C) Alternativa incorreta.

Justificativa: o erro está em afirmar que um processo não pode executar em sua própria seção crítica.

D) Alternativa incorreta.

Justificativa: a correlação com carros de verdade realizando malabarismo é absurda.

E) Alternativa incorreta.

Justificativa: a seção crítica de um processo não deve ser compartilhada de forma indiscriminada.

Questão 2. Os alunos do curso de computação de uma grande universidade utilizam a linguagem de programação C para implementar seus programas. A figura 1 ilustra os procedimentos realizados pelo compilador da linguagem C para gerar um arquivo executável que, em um primeiro momento, é apenas uma entidade passiva, ocupando *bytes* da memória secundária. A fim de que essa entidade passiva tenha utilidade, precisa ser transformada em um processo. Todo processo, para ser executado, necessita estar adicionado à fila de processos prontos.

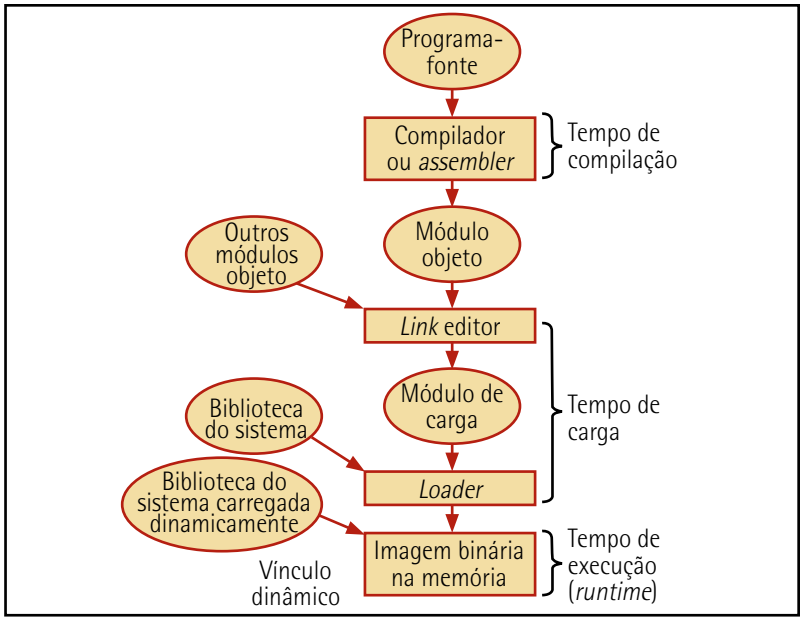


Figura A – Procedimentos realizados pelo compilador da linguagem C (SILBERSCHATZ, GALVIN e GAGNE, 2007)

Durante a execução de um processo, são utilizados, em geral: variáveis, objetos, funções etc. que deverão ser alocados na memória. A Figura B ilustra a MMU em operação.

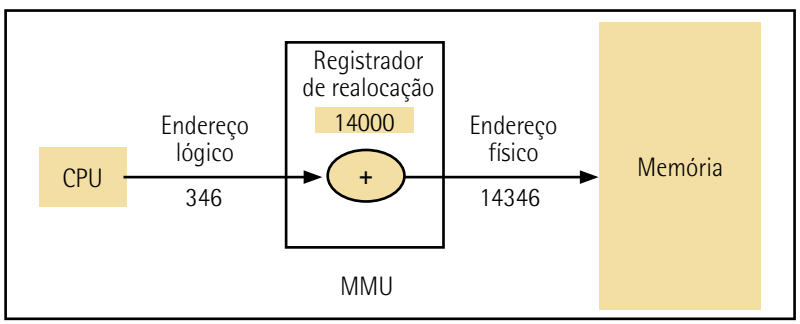


Figura B – MMU em operação (SILBERSCHATZ, GALVIN e GAGNE, 2007)

Qual a função da MMU no desenvolvimento de execução de processo?

A) Converter um endereço físico em um endereço lógico.

B) Dispositivo de *hardware* que mapeia endereços virtuais em físicos.

C) Adicionar o endereço lógico à memória.

D) Servir de registrador de realocação entre a CPU e a memória.

E) Monitorar os procedimentos dos programas na CPU e na memória.

Resolução desta questão na Plataforma.