O Sistema Operacional que você usa é *multitasking*?

Por ***multitasking***, entende-se a capacidade do SO de ter mais de um processos em execução ao mesmo tempo.

É claro que, num dado instante, o número de processos que pode estar tendo suas instruções executadas é limitado ao número de processadores, ou cores (núcleos), disponíveis no computador. Assim, se há apenas 1 processador com um único núcleo, num instante qualquer esse processador vai estar executando instruções de um único processo. Se houver 2 núcleos, 2 processos podem estar tendo suas instruções executadas nesse instante e, assim, sucessivamente.

Entretanto, Sistemas Operacionais comumente possuem um mecanismo para salvar o estado de execução de um processo, o que corresponde ao seu contexto. Tendo salvo essas informações, que refletem exatamente quais eram os valores nos registradores do processador, é possível restaurá-las posteriormente, de forma que a execução do processo interrompido pode prosseguir como se ela não tivesse sido interrompida.

Quando o SO salva o contexto de um processo?

O salvamento de contexto em geral ocorre quando há uma interrupção. São as interrupções que permitem ao SO retomar o controle de um processador.

A estrutura interna de um SO é geralmente um pouco diferente dos processos comuns e ele tem certos privilégios de execução. Contudo, o SO também é um programa, assim como os programas de usuário, e suas ações também são escritas na forma de sequências de instruções. Assim, se o processador foi atribuído para a execução das instruções de um processo de usuário, o SO não tem como recuperar o uso do processador, a não ser que algum evento ocorra. Esse evento é uma interrupção.

Interrupções ocorrem independentemente da ação do Sistema Operacional e podem ser síncronas ou assíncronas. Interrupções **assíncronas** são geradas por controladores de dispositivos, como o controlador de disco, ou a interface de rede, e indicam que houve alterações nas operações desses dispositivos que requerem a atenção do SO.

O controlador presente na interface de rede, por exemplo, pode gerar uma interrupção ao processador para indicar que um novo pacote de rede destinado a este computador foi recebido, que sua consistência foi verificada e este pacote está correto e precisa ser tratado. Agora é preciso que o SO repasse esse pacote para o módulo tratador do protocolo de rede apropriado. Para conseguir chamar a atenção do SO, o controlador de rede gera uma interrupção.

Interrupções **síncronas** são geradas em decorrência da execução de instruções. Elas ocorrem quando há algum problema na execução de uma instrução, como uma tentativa de divisão por 0, ou um acesso a uma posição inválida de memória.

Outro tipo de interrupção síncrona é a **instrução** de interrupção (*int*), presente em arquiteturas x86. A execução dessa instrução gera o que é chamado de *software interrupt* e é usada para os programas solicitarem serviços do SO. Em alguns SOs, instruções específicas dos processadores modernos também podem ser usadas para a chamada dos serviços do SO, como *sysenter*, para processadores Intel, e *syscall*, para processadores AMD.

Quando uma interrupção ocorre, independentemente do seu tipo, o **processador**[1] salva

1 É importante perceber que é o processador que salva o ponteiro de instruções na pilha, e não o SO. Se fosse o SO, isso significaria que ele estaria salvando o endereço da sua próxima instrução, e não do processo que estava em execução antes que a rotina de tratamento de interrupção entrasse em execução.

automaticamente na pilha, em memória, o conteúdo do registrador que indica qual é a próxima instrução a executar para o processo corrente nesse processador. O número da interrupção é então usado como um índice num vetor de endereços de rotinas de tratamento de interrupções e o endereço presente na posição indicada é atribuído ao registrador que aponta para a próxima instrução a executar. Como o SO fez fez o ajuste do conteúdo desse vetor, colocando ali os endereços de suas rotinas apropriadas para cada caso, é o próprio SO que passa à execução quando uma interrupção ocorre.

Assim, o SO retoma o controle do processador (ou de um dos processadores) sempre que uma interrupção ocorre. Essa interrupção pode estar relacionada com o processo que estava executando, seja porque um *trap* ocorreu durante a execução, ou porque esse processo solicitou um serviço do SO, ou pode ter sido gerada por algum dispositivo, como o *timer* ou um controlador de dispositivo de E/S. Fica a critério do SO, então, decidir se o processo que estava em execução naquele processador deve ter seu contexto salvo e ser movido de volta para a fila de prontos, ou não.

De maneira resumida, uma troca de contexto pode ocorrer sempre que o SO retoma o controle de um processador. Caso o processo em execução não faça chamadas de serviço ao SO, ou nenhum dispositivo de entrada e saída de dados que estava realizando serviço gere uma interrupção, o SO sempre conta com as interrupções do dispositivo de *timer* para retomar o controle periodicamente. Quando uma interrupção do *timer* ocorre, isso indica ao SO que a fatia de tempo do processo que estava em execução terminou.

Gerenciando múltiplos processos (tarefas, ou *tasks*)

A técnica de **multiprogramação**, que sobrepõe serviços de entrada e saída de dados dos processos solicitantes com a execução de outros, e o uso de **fatias de tempo** são, então, os principais mecanismos usados por um SO multitarefa para promover as trocas de contexto. Isso permite que existam mais processos em execução do que o número de processadores disponíveis num computador.

Quando um programa solicita ao usuário que digite algo, esse programa não pode prosseguir sua execução enquanto não tiver os dados que solicitou. É possível fazer um programa que, passado algum tempo sem receber os dados solicitados, não fique indefinidamente bloqueado nessa espera. De maneira geral, contudo, os programas ficam parados até que o SO consiga obter os dados solicitados e os repasse a esses programas.

O mesmo vale quando um programa quer ler dados de um arquivo. Uma transferência desse tipo envolve: verificar se os dados solicitados já estão na memória e, caso não estejam, alocar um espaço na memória para colocá-los; localizar onde esses dados estão no disco; solicitar ao controlador do dispositivo de armazenamento a transferência de um ou mais blocos de dados para a memória (essa etapa demora um tempo razoável); copiar os dados recebidos do controlador para a posição de memória indicada pelo programa; e retomar a execução do programa.

A etapa de transferência dos dados é, em geral, realizada de maneira **autônoma** pelo controlador do dispositivo, que copia os dados obtidos do disco, por exemplo, para a posição de memória indicada pelo SO. Isso pode ser feito usando técnicas de acesso direto à memória (DMA), sem que o processador tenha que ler cada *byte* (ou palavra) do controlador. Ao invés de ficar ocioso durante esse período, o SO salva o contexto do processo que aguarda essa transferência, seleciona um novo processo para executar, restaura o seu contexto no hardware e transfere o uso do processador para esse processo.

O SO não precisa preocupar-se em retomar o controle pois isso irá ocorrer em breve. Se o processo agora em execução precisar serviços do SO, o chama. Se algum dispositivo precisar a atenção do SO, gera uma interrupção externa. Se nenhum desses casos ocorrer, o dispositivo de *timer* gera uma interrupção externa no tempo máximo definido pelo SO como fatia de tempo para execução de cada processo numa rodada.

Pronto! Assim, um SO pode ser multitarefa e, ao longo do tempo, vários processos vão sendo executados. Com processadores rápidos existentes atualmente, usuários geralmente nem percebem que o uso da CPU está sendo alternado entre diversos processos. A impressão que se tem é que todos os processos possuem processadores dedicados para suas execuções.

Sistema Operacional multiusuário

Suportando múltiplos processos, também é possível ao Sistema Operacional ser **multiusuário**. Sistemas multiusuários dão a ideia de computadores com múltiplos terminais de acesso, com teclado e monitor independente para cada usuário. Quando se trata de um computador pessoal (PC), com um único teclado, mouse e apenas um monitor de saída, contudo, pode parecer estranho pensar-se em suporte a vários usuários.

Com a possibilidade de interligação em rede, é possível que usuários remotos, que interagem fisicamente com outros computadores e seus respectivos dispositivos de digitação e visualização, até mesmo um computador pessoal pode ser considerado multiusuário. Sistemas Operacionais atuais em versões de servidor permitem o acesso aos seus recursos por usuários remotos. Servidores de terminal (*terminal servers*) e de *shell* remoto (*ssh*, por exemplo) são serviços que podem ser ativados num servidor para permitir o atendimento simultâneo a múltiplos usuários. O compartilhamento dos recursos desse computador, como os processadores disponíveis, pode não fazer distinção entre processos de um usuário local e de usuários remotos.

Multiprocessamento: suporte para mais de um processadores ou cores

Quando há mais de um processadores disponíveis, sejam eles chips separados na *motherboard,* ou múltiplos *cores* num mesmo chip, temos um computador **multiprocessado.** Se o Sistema Operacional identifica e gerencia todos esses processadores, temos um SO com suporte a multiprocessamento.

A lógica de divisão do uso dos processadores é a mesma do caso em que há um único processador. O SO até poderia alocar um processador dedicado para suas atividades, deixando os demais para executar os processos de usuários. Isso não ocorre, contudo, uma vez que o SO deve ser rápido, eficiente, e usar os processadores o mínimo possível, maximizando o tempo em que esses processadores estão disponíveis para executar processos de usuário.

Para que servem múltiplas *threads* num mesmo processo?

Threads são **linhas de execução** associadas a um processo. Quando há múltiplas *threads* associadas a um processo, é comum que diferentes funções desse processo sejam tratadas como se fossem processos. Também é possível que existam múltiplas instâncias de uma mesma função desse processo, todas executando o mesmo código mas, normalmente, manipulando partes diferentes dos

dados que precisam ser tratados pelo processo.

Se o programador não tratar explicitamente do uso de *threads,* normalmente cada processo tem apenas uma única *thread.* Para programas escritos na linguagem C, essa *thread* está associada à função *main( )*.

Num computador com um único processador (com um único núcleo), apenas 1 processo pode estar em execução num dado instante. Será que um processo pode beneficiar-se do uso de várias *threads* nesse caso?

Sim.

Por exemplo, podemos pensar um editor de texto. Por mais rápida que seja a capacidade de digitação do usuário, é comum que haja momentos de pausa na digitação. Nesse caso, ao invés de ficar completamente parado, o processo poderia usar uma *thread* para esperar os dados digitados e outra *thread* para fazer uma cópia temporária no disco, ou ainda realizar a verificação ortográfica.

Com um navegador WWW, o raciocínio é o mesmo. Enquanto uma *thread* pode estar ocupada esperando se o usuário digita uma nova URL (e.g. *http://ead.ufscar.br/*), outras *threads* poderiam estar em execução, buscando diferentes partes de uma outra URL, digitada anteriormente ou selecionada com o mouse, formatando a página para exibição, apresentando animações ou vídeos, etc.

No caso de um processo servidor WWW, o benefício do uso de várias *threads* é ainda mais evidente, pensando que um mesmo processo servidor poderia estar atendendo diferentes requisições de diferentes clientes simultaneamente. Enquanto uma *thread* é bloqueada à espera de dados solicitados, outras *threads* podem estar buscando outros dados, formatando dados para envio, processando requisições, etc.

De maneira resumida, quando há apenas um processador, o uso de *threads* é interessante para **evitar bloqueios** e para sobrepor a execução de instruções do processo com suas próprias operações de entrada e saída de dados.

Quando há vários processadores ou cores, o uso de várias *threads* num processo é vantajoso também por outros motivos. Se um processo tem uma única *thread,* apenas um processador será alocado para sua execução. Esse processo pode ser tratado ora por um processador, ora por outro, mas apenas por um de cada vez.

Se o processo tiver mais *threads*, havendo processadores disponíveis, mais de um deles podem estar executando *threads* desse processo ao mesmo tempo. Supondo que há 2 processadores no computador, ou um processador com 2 cores (núcleos), se o processo tem 2 *threads,* cada *core* pode fazer **metade** do serviço **ao mesmo tempo**.

Assim, quando o volume de processamento é grande, geralmente é possível dividir essas atividades em partes e usar 1 processador para cada uma delas. Supondo que tivéssemos 4 cores, seria possível executar o programa em aproximadamente 1/4 do tempo. Se tivéssemos um número muito grande de processadores, 1000, por exemplo, e todo o programa pudesse ser dividido de forma que cada parte realizasse uma fração igual das atividades do processo, teoricamente, o tempo de execução seria 1/1000 (um milésimo) do tempo total.

É claro que há atividades num processo que não podem ser executadas em paralelo. Há estudos que mostram que o aumento máximo de desempenho que um programa pode obter com o paralelismo está limitado pelo tamanho de sua parte que tem que ser executada sequencialmente.

Múltiplos processos x múltiplas *threads*

É certo que o atendimento de várias requisições num processo servidor poderia ser realizado na

forma de múltiplos processos. Também a paralelização do processamento de uma aplicação, por exemplo, o processamento de uma imagem de forma dividida, pode ser feito usando múltiplos processos na mesma aplicação. Fazer isso com *threads,* contudo, é muito mais simples do que com processos.

Threads compartilham a maior parte das áreas de memória de um processo. Também é possível compartilhar as áreas de dados entre processos. Isso é feito embutindo no programa chamadas explícitas ao SO com indicações das áreas que serão compartilhadas. Todo o ajuste de ponteiros para essas áreas deve ser feito pelo programador, contudo.

Esse compartilhamento é muito mais simples e eficiente com *threads.*

Considerando a área de código de um processo, ela é naturalmente compartilhada entre suas *threads*. Essa área de código, contudo, não é compartilhada entre processos e não há mecanismos para fazer isso explicitamente. É certo que alguns Sistemas Operacionais possuem otimizações para evitar cópias de memória e para compartilhar certas áreas de memória entre processos pai e filhos, mas isso é uma otimização interna desses SOs, que não está ao alcance do controle dos processos.

Deste modo, várias questões favorecem o uso de múltiplas *threads* ao invés de múltiplos processos na mesma aplicação: compartilhamento (e economia) de recursos, facilidade de utilização e desempenho.