Processos

A CPU executa somente um programa, no decorrer de um segundo ela pode trabalhar sobre vários programas, dando aos usuários a ilusão de paralelismo.

Modelo de processo: Um processo é apenas um programa em execução acompanhado dos valores atuais do *contador de programa (PC)*, dos registradores e das variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. Esse mecanismo de trocas rápidas é chamado de *multiprogramação*, porém há somente um PC físico. Assim, quando cada processo executa, seu PC lógico é carregado no PC real. Terminando o tempo de CPU alocado para um processo, o PC físico é salvo no PC lógico do processo na memória. Um modo de ver um processo é encará-lo como um meio de agrupar recursos relacionados.

Com a alternância da CPU entre os processos, a taxa na qual o processo realiza sua computação não será uniforme e provavelmente não será nem reproduzível se os mesmos processos executarem novamente. Desse modo os processos não devem ser programados com hipóteses predefinidas sobre a temporização.

Criação de processos: Os sistemas operacionais (SO) precisam assegurar de alguma modo a existência de todos os processos necessários. Há quatro eventos principais que fazem com que processos sejam criados:

1. Início do sistema.
2. Execução de uma chamada ao sistema de criação de processo por um processo em execução.
3. Uma requisição do usuário para criar um novo processo.
4. Início de um job em lote.

Quando um SO é carregado, em geral criam-se vários processos. Alguns deles são processo em primeiro plano, ou seja, que interagem com usuários (humanos) e realizam tarefas para eles. Processos que ficam em segundo plano com a finalidade de tratar alguma atividade como mensagem eletrônica, páginas Web, noticias, impressão, entre outros, são chamado de ***deamons***. Um processo em execução emitirá chamadas ao sistema para criar um ou mais novos processos para ajudá-lo em seu trabalho. Criar novos processos é particularmente útil quando a tarefa a ser executada pode facilmente ser formulada com base em vários processos relacionados, mas interagindo de maneira independente.

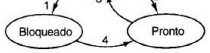
O que o processo faz é executar uma chamada ao sistema para criar um novo processo e assim indica, direta ou indiretamente, qual programa executar nele. A chamada *fork* cria um clone idêntico ao processo que o chamou. Depois da *fork*, os dois processos, o pai e o filho, têm a mesma imagem de memória, as mesmas cadeias de caracteres no ambiente e os mesmos arquivos abertos. O *execve* ou uma chamada similar ao sistema para mudar sua imagem de memória e executar um novo programa. Tanto no *Unix* quanto no *Windows*, depois que um processo é criado, o pai e o filho têm seus próprios e distintos espaços de endereçamento. Se um dos dois processos alterarem uma palavra em seu espaço de endereçamento, a mudança não será visível ao outro processo.

Término e hierarquia de processos: Motivos básicos para o término de processos:

1. Saída normal (voluntário).
2. Saída por erro (voluntário).
3. Erro fatal (involuntário).
4. Cancelamento por um outro processo (involuntário).

Em alguns sistemas, quando um processo cria outro processo, o processo pai e o processo filho continuam, de certa maneira, associados. O próprio filho pode gerar mais processos, formando uma hierarquia de processos.

Estados de processos: Um processo pode gerar uma saída que outro processo usa como entrada. Veja a máquina de estado e a descrição para os estados e transições:



Estados:

1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável; temporariamente parado para dar lugar a outro processo).
3. Bloqueado (incapaz de executar enquanto um evento externo não ocorrer).

Transições:

1. O processo bloqueia aguardando uma entrada.
2. O escalonador seleciona outro processo.
3. O escalonador seleciona esse processo.
4. A entrada torna-se disponível.

A transição 2 e 3 são causadas pelo escalonador de processos sem que o processo saiba disso. A transição 2 ocorre quando o escalonador decide que o processo em execução já teve tempo suficiente de CPU e é momento de deixar outro processo ocupar o tempo da CPU. A transição 3 ocorre quando todos os outros processos já compartilharam a CPU, de uma maneira justa, e é hora de o primeiro processo obter novamente a CPU. A transição 4 ocorre quando acontece um evento externo pelo qual um processo estava aguardando (como a chegada de alguma entrada). Se nenhum outro processo estiver executando naquele momento, a transição 3 será disparada e o processo executará. Caso contrário, ele poderá ter de aguardar em estado de *pronto* por um pequeno intervalo de tempo, até que a CPU esteja disponível e sua vez chegue.

Implementação de processos: O SO mantém, uma tabela chamada de ***tabela de processos***, com uma entrada para cada processo. Essa entrada contém informações sobre o estado do processo, seu PC, o ponteiro da pilha, a alocação de memória, os estados de seus arquivos abertos, e tudo o mais sobre o processo que deva ser salvo quando o processo passar do estado *em execução* para o estado *pronto* ou *bloqueado*, para que ele possa ser reiniciado depois, como se nunca tivesse sido bloqueado.

Associada a cada classe de dispositivos de *I/O* está uma locação de memória (geralmente próxima da parte mais baixa da memória) chamada de ***vetor de interrupção***. Esse vetor contém os endereços dos procedimentos dos serviços de interrupção.

Todas as interrupções começam salvando os registradores, muitas vezes na entrada da tabela de processo do processo atual. Então a informação colocada na pilha pela interrupção é removida.

Segue o resumo do tratamento de interrupções:

1. O hardware empilha o PC etc.
2. O hardware carrega o novo PC a partir do vetor de interrupção.
3. O procedimento em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.

Threads

Cada processo tem um espaço de endereçamento e um único fluxo (*thread*) de controle. Este é um conceito a mais sobre processos e ***não*** algo novo e desconexo.

Modelo de Thread: O *thread* tem um PC que mantém o controle de qual instrução ele deve executar em seguida. Ele tem registradores que contêm suas variáveis atuais de trabalho. Apresenta uma pilha que traz a história de execução, com uma estrutura para cada procedimento chamado mas ainda não retornado.

Threads são entidades escalonadas para a execução sobre a CPU. O que os *threads* acrescentam ao modelo de processo é permitir que múltiplas execuções ocorram no mesmo ambiente do processo com um grande grau de independência uma da outra. Te múltiplos *threads* executando em paralelo em um processo é análogo a múltiplos processo executando em paralelo em um único computador. O termo *multithread* é também usado para descrever a situação em que se permite a existência de múltiplos *threads* no mesmo processo.

Quando um processo com múltiplos *threads* é executado em um sistema com uma úncia CPU, os *threads* esperam esperam a vez para executar. A CPU alterna rapidamente entre os *threads* dando a impressão de que os *threads* estão executando em paralelo.

Threads distintos em um processo não são tão independentes quanto processo distintos. Todos os *threads* têm exatamente o mesmo espaço de endereçamento, o que significa que eles também compartilham as mesmas variáveis globais. Como cada *thread* pode ter acesso a qualquer endereço de memória dentro do espaço de endereçamento do processo, um *thread* pode ler, escrever ou até mesmo apagar completamente uma pilha de outro *thread*. Não há proteção entre *threads* porque (1) é impossível e (2) não seria necessário.

Um *thread* pode estar em vários estados: em execução, bloqueado, pronto ou finalizado. Além disto em execução ele detém a CPU e quando bloqueado fica esperando por algum evento que o ative novamente. Um *thread* pronto está escalonado para executar e logo se tornará ativo. É importante perceber que cada *thread* tem sua própria pilha. Cada pilha de *thread* contém uma estrutura para cada procedimento chamado, mas que ainda não retornou.

Cada *thread* geralmente chama procedimentos diferentes resultando uma história de execução diferente. Por isso é que o *thread* precisa ter sua própria pilha. Quando ocorre a execução de múltiplos *threads*, os processos normalmente inciam com um único *thread*.

Mesmo sendo úteis em muitas situações, os *threads* também introduzem várias complicações no modelo de programação. Só para começar, considere os efeitos da chamada ao sistema fork do Unix.

Uso de thread: A principal razão para existirem *threads* é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. Algumas dessas atividades podem bloquear com o tempo. O modelo de programação se torna mais simples se decompormos uma aplicação em múltiplos *threads* sequenciais que executam em quase paralelo.

Só que agora, com os *threads*, adicionamos a capacidade de entidades paralelas compartilharem de um espaço de endereçamento e todos os seus dados entre elas mesmas.

Threads são mais fáceis de criar e destruir que os processos, pois não têm quaisquer recursos associados a eles.

O uso de *threads* não resulta em ganho de desempenho quando todos eles são orientados à CPU. No entanto, quando há grande quantidade de computação e de *I/O*, os *threads* permitem que essas atividades se sobreponham e, desse modo, aceleram a aplicação.

Threads tornam possível manter a ideia de processo sequenciais que fazem chamadas ao sistema com bloqueio e mesmo assim conseguem obter paralelismo.

Implementação de *Threads* de usuário: O primeiro método é inserir o pacote de *thread* totalmente dentro do espaço do usuário (*thread* de usuário). O núcleo não é informado sobre eles. O que compete ao núcleo é o gerenciamento comum de processos *monothread*. A primeira vantagem e a mais óbvia é que um pacote de *threads* de usuário pode ser implementado em um SO que não suporta *threads*.

Cada processo precisa de sua própria tabela de *threads* para manter o controle dos *threads* naquele processo. Essa tabela é análoga à tabela de processos do núcleo, exceto por manter o controle apenas das propriedades do *thread*, como o PC, o ponteiro para pilha …

Quando um *thread* faz algo que possa bloqueá-lo localmente, ele chama um procedimento do sistema supervisor. Esse procedimento verifica se o *thread* deve entrar no estado bloqueado. Em caso afirmativo, ele armazena os registradores do *thread* (isto é, os seus próprios) na tabela de *threads*, busca na tabela por um *thread* pronto para executar e recarrega os registradores da máquina com novos valores salvos do *thread*. Logo que o ponteiro de pilha e o PC forem alterados, o novo *thread* reviverá automaticamente.

Fazer assim a alternância de *threads* é, pelo menos, de uma ordem de magnitude mais rápida que desviar o controle para o núcleo – um forte argumento em favor do thread de usuário.

Thread de usuário têm também outras vantagens, por exemplo, permitem que cada processo tenha seu próprio algoritmo de escalonamento personalizado. Eles também escalam melhor, já que os *threads* de núcleo invariavelmente necessitam de algum espaço de tabela e espaço de pilha no núcleo, o que pode vir a ser um problema caso haja um número muito grande de *threads*.

Threads de usuários também apresentam algumas desvantagens como: Problema da chamada ao sistema, falta de pagina e *Threads* executando indefinidamente.

Implementação de Threads de núcleo: A tabela de *threads* do núcleo contém os registradores, o estado e outras informações de cada *thread*. As informações são as mesmas dos *threads* de usuário, mas estão agora no núcleo, e não no espaço do usuário. O núcleo também mantém a tabela de processos para acompanhamento dos processos.

Todas as chamadas que possam bloquear um *thread* são implementadas como chamadas ao sistema, com um custo consideravelmente maior que uma chamada para procedimento do sistema supervisor. Quando um *thread* é bloqueado, é opção do núcleo executar outro *thread* do mesmo processo ou um *thread* de outro processo.

Por causa do custo relativamente maior de criar e destruir *threads* de núcleo, alguns sistemas adotam uma abordagem 'ambientalmente correta' e 'recicla' seus *threads*. Ao ser destruído, um *thread* é marcado como não executável, mas suas estruturas de dados no núcleo não são afetadas.

A principal desvantagem é que o custo de uma chamada ao sistema é alto e, portanto, a ocorrência frequentemente de operações de *thread* causará uma sobrecarga muito maior.

Implementações hibridas: Núcleo sabe apenas sobre *threads* de núcleo e escalona-os.

Ativações do escalonador: A eficiência é conseguida evitando-se transições desnecessárias entre o espaço do usuário e o do núcleo. Por exemplo, se um *thread* bloqueia aguardando que outro *thread* faça algo, não há razão para envolver o núcleo, economizando assim a sobrecarga da transição núcleo usuário. O sistema supervisor no espaço do usuário pode bloquear o *thread* de sincronização e ele mesmo escalar outro.

Quando o núcleo sabe que um *thread* bloqueou, ele avisa o sistema supervisor do processo, passando, como parâmetro na pilha, o número do *thread* em questão e uma descrição do evento ocorrido. A notificação ocorre quando o núcleo ativa o sistema supervisor em um endereço inicial conhecido, semelhante a um sinal no Unix.

Uma vez ativado, o sistema supervisor pode reescalonar seus *threads*, geralmente marcando o *thread* atual como bloqueado e tomando outro *thread* da lista de prontos, configurando seus registradores e reiniciando-o. Depois, quando o núcleo souber que o *thread* original pode executar novamente, o núcleo faz um outro ***upcall*** para o sistema supervisor para informá-lo sobre esse evento. O sistema supervisor, por conta própria, pode reiniciar o thread bloqueado imediatamente ou colocá-lo na lista de pontos para executar adiante.

Quando ocorre uma interrupção de hardware, enquanto um *thread* de usuário estiver executando, a CPU interrompida vai para o modo núcleo. Quando o tratador da interrupção termina, ele colocará o *thread* interrompido de volta no estado em estava antes da interrupção. Se, contudo, o processo estiver interessado na interrupção, o *thread* interrompido não será reiniciado. Em vez disso, o *thread* interrompido será suspenso e o sistema supervisor será inciado sobre a CPU virtual, com o estado do *thread* interrompido presente na pilha.