1.Modelo de processos

Todo o software executável no computador, às vezes incluindo o sistema operacional, é organizado em diversos ***processos sequenciais***, ou apenas ***processos***. Um processo é simplesmente um programa em execução, incluindo os valores correntes do contador de programa, dos registradores e das variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. A CPU alterna de um processo para outro, também conhecido como ***multiprogramação***.

Existe apenas um contador de programa físico, de modo que, quando cada processo é executado, seu contador de programa lógico é carregado no contador de programa físico. Quando ele termina, o contador de programa físico é salvo no contador de programa lógico do processo em memória.

Com a CPU alternando entre os processos, a velocidade com que um processo faz sua computação não será uniforme e, provavelmente, nem mesmo poderá ser reproduzida se os mesmos processos forem executados novamente. Assim, os processos não devem ser programados com suposições sobre temporização estabelecidas. Quando um processo tem requisitos de tempo real crítico, medidas especiais devem ser tomadas para garantir que eles sejam cumpridos.

A diferença entre um processo e um programa é sutil. Considere um profissional de computação com dotes culinários que está assando um bolo de aniversário para a sua filha. Ele tem uma receita de bolo e uma cozinha bem equipada, com todos os ingredientes necessários. Neste caso, a receita de bolo é o programa (algoritmo), o profissional de computação é o CPU e os ingredientes do bolo são os dados de entrada. O processo é a atividade que consiste no confeiteiro ler a receita, buscar os ingredientes e assar o bolo.

Agora imagine que algum empecilho acabou surgindo e o confeiteiro teve que parar de fazer o bolo. O profissional memoriza o ponto onde estava na receita (estado do processo) e resolve o que precisa ser resolvido. Aqui vemos o processador alternando de um processo para o outro, com prioridade mais alta/baixa. Nesta analogia, a ideia-chave é que um processo é um tipo de atividade. Ele tem um programa de entrada, saída e um estado. Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de escalonamento sendo utilizado para determinar quando deve interromper o trabalho em um processo e atender outro.

1.1 Criação de processos

Os sistemas operacionais precisam de alguma maneira de garantir que todos os processos necessários existam. Existem quatro eventos principais que acarretam a criação de processos:

- Inicialização do sistema
- Realização de uma chamada de sistema por um processo em execução para criação de processo
- Um pedido de usuário para criar um novo processo

- Início de uma tarefa em lote

Quando um sistema operacional é inicializado, frequentemente vários processos são criados. Estes processos podem ser de *foreground* ou de *background*. Os de *foreground* interagem com os usuários e executam trabalho para eles. Os de *background* não são associados a usuários em particular, mas tem alguma função específica.

Os processos que ficam em segundo plano para executar alguma atividade, como buscar páginas web, impressão, entre outros, são chamados de ***daemons***. Os sistemas grandes normalmente têm dezenas deles.

Frequentemente, um processo em execução fará chamadas de sistema para criar um ou mais processos novos, para ajudá-lo a fazer seu trabalho. A criação de novos processos é particularmente útil quando o trabalho a ser feito pode ser facilmente formulado em termos de vários processos relacionados que estão interagindo, mas que são independentes.

Nos sistemas interativos, os usuários podem iniciar um programa digitando um comando. No MINIX 3, consoles virtuais permitem que um usuário inicie um programa, e depois troque para um console alternativo e inicie outro programa.

A última situação onde processos são criados se aplica apenas aos sistemas de lote encontrados nos computadores de grande porte. Aqui, os usuários podem submeter tarefas de lote para o sistema. Quando o sistema operacional decide que tem recursos suficientes para executar outra tarefa, ele cria um novo processo e executa a próxima tarefa de sua fila de entrada.

Em todos esses casos, um novo processo é criado fazendo-se com que um processo existente execute uma chamada de sistema para criação de processo. Esse processo pode ser um processo de usuário em execução, um processo de sistema ativado a partir do teclado ou mouse, ou ainda um processo do gerenciador de lotes. O que esse processo faz é executar uma chamada de sistema para criar o novo processo.

No MINIX 3, existe apenas uma chamada de sistema para criar um novo processo

FORK

Essa chamada cria um clone exato do processo que fez a chamada. Após, os dois processos, o pai e o filho, têm a mesma imagem da memória, as mesmas strings de ambiente e os mesmos arquivos abertos. Isso é tudo.

No MINIX 3 e no UNIX, depois que um processo é criado, tanto pai quanto filho, eles têm seus próprios espaços de endereçamento distintos. Se um dos processos alterar uma palavra em seu espaço de endereçamento, ela não será visível para o outro processo. O espaço de endereçamento inicial do filho é uma cópia do espaço de endereçamento do pai, mas existem dois espaços de endereçamento distintos envolvidos. Nenhuma porção de memória passível de ser escrita é compartilhada. Entretanto, é possível que um processo recentemente criado compartilhe alguns outros recursos de seu criador, como os arquivos abertos.

1.2 Término de processos

Após um processo ser criado, ele começa a ser executado e faz seu trabalho, seja qual for. Uma hora ou outra este processo será terminado devido a uma das seguintes condições.

- Término voluntário
- Término por erro (voluntário)
- Erro fatal (involuntário)
- Eliminado por outro processo (involuntário)

A maioria dos processos terminam porque já fez o seu trabalho. Quando um compilador tiver compilado o programa recebido, ele executa uma chamada de sistema para dizer ao sistema operacional que terminou. No MINIX 3, essa chamada é a *exit*.

Os programas também aceitam término voluntário.

O segundo motivo de término é o fato do processo descobrir um erro fatal. Se um usuário digita um comando para compilar determinado programa e esse arquivo não existir, o compilador simplesmente encerrará.

O terceiro motivo para o término é o erro causado pelo processo, talvez devido a um erro no programa. Exemplos incluem a execução de uma instrução inválida, referência à memória inexistente ou divisão por 0. No MINIX 3, um processo pode dizer ao sistema operacional que deseja tratar de certos erros sozinho, no caso em que o processo é sinalizado (interrompido), em vez de terminar quando um dos erros ocorre.

O quarto motivo é o fato de executar uma chamada de sistema instruindo o sistema operacional a eliminar algum outro processo. No MINIX 3, essa chamada é *kill*. É claro que o processo que vai eliminar o outro deve ter a autorização necessária para isso. Em alguns sistemas, quando um processo termina, voluntariamente ou não, todos os processos que criou também são eliminados imediatamente. No MINIX 3 não funciona assim.

1.4 Hierarquia de processos

O próprio processo filho pode criar mais processos, formando uma hierarquia de processos. Um processo tem apenas um pai, porém, diversos ou nenhum filho.

No MINIX 3, um processo, seus filhos e outros descendentes podem, juntos, formar um grupo de processos. Quando um usuário envia um sinal do teclado, o sinal pode ser enviado para todos os membros do grupo de processos corretamente associados ao teclado. Isso é a dependência de sinal.

Se um sinal é enviado para um grupo, cada processo pode capturá-lo, ignorá-lo ou executar a ação padrão, que é ser eliminado pelo sinal.

Como um exemplo simples de como as árvores de processos são utilizadas, será mostrado como o MINIX 3 se inicializa. Dois processos especiais, o ***servidor de reencarnação*** e ***init*** estão presentes na imagem de *boot*. A tarefa do servidor de reencarnação é (re)iniciar *drivers* e servidores. Ele começa bloqueado, à espera de mensagens que o instrua sobre o que criar.

Em contraste, *init* executa o *script /etc/rc*, que o faz enviar comandos para o servidor de reencarnação para iniciar os *drivers* e servidores ausentes na imagem de *boot*. Esse procedimento torna os *drivers* e os servidores filho do servidor de reencarnação, de modo que, se qualquer um deles terminar, o servidor de reencarnação será informado e poderá reiniciá-los (reencarna-los) novamente.

Esse mecanismo se destina a permitir que o MINIX 3 tolere uma falha de *driver* ou de servidor, pois um novo *driver* ou servidor será iniciado automaticamente. Contudo, na prática, substituir um *driver* é muito mais fácil do que substituir um servidor, pois há menos repercussão em outras partes do sistema.

Quando *init* tiver terminado de fazer isso, ele lê um arquivo de configuração (/etc/ttytab) para ver quais terminais reais e virtuais existem. *Init* cria com *fork* um processo *getty* para cada um deles, exibe um prompt de *login* e depois espera pela entrada. Quando um nome é digitado, *getty* executa um processo *login* tendo o nome como seu argumento. Se o usuário tiver êxito na conexão, *login* executará o *shell* do usuário. Portanto, o *shell* é um filho de *init*. Essa sequência de eventos é um exemplo de como as árvores de processos são usadas.

1.5 Estados de um processo

Embora cada processo seja uma entidade independente, com seu próprio contador de programa, registradores, pilha, arquivos abertos, alarmes e outros estados internos, os processos frequentemente precisam interagir, se comunicar e se sincronizar com outros processos.

Pode ser que, em determinado processo, determinada parte esteja pronta para executar, mas não haja nenhuma entrada esperando por este processo. Este processo então deve ser bloqueado até que esta entrada esteja disponível.

Quando um processo é bloqueado, isso acontece porque logicamente ele não pode continuar, normalmente, porque está esperando uma entrada que ainda não está disponível. Também é possível que um processo que esteja conceitualmente pronto e capaz de executar, seja interrompido porque o sistema operacional decidiu alocar a CPU temporariamente para outro processo.

Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema, já no segundo, trata-se de um aspecto técnico do sistema.

Em alguns sistemas, o processo precisa executar uma chamada de sistema *block* ou *pause*, para entrar no estado bloqueado. Em outros sistemas, incluindo o MINIX 3, quando um processo lê um *pipe* ou um arquivo especial e não há nenhuma entrada disponível, ele muda automaticamente do estado em execução para o estado bloqueado.

Usando o modelo de processos, torna-se muito mais fácil pensar no que está ocorrendo dentro do sistema. Alguns processos executam programas que executam comandos digitados por um usuário. Outros processos fazem parte do sistema e executam tarefas como fazer requisições de serviços de arquivo ou gerenciar os detalhes da operação de um disco ou de uma unidade de fita.

Quando ocorre uma interrupção de disco, o sistema pode tomar a decisão de parar de executar o processo corrente e executar o processo de disco, que estava bloqueado esperando essa interrupção. Quando o bloco de disco for lido ou o caractere digitado, o processo que estava esperando por isso é desbloqueado e ele fica pronto para executar novamente.

1.6 Implementação de processos

Para implementar o modelo de processos, o sistema operacional mantém uma tabela chamada ***tabela de processos***, com uma entrada por processo. Essa entrada contém informações sobre o estado do processo, sobre seu contador de programa, sobre o ponteiro da pilha, sobre a alocação de memória, sobre o status de seus arquivos abertos, suas informações de contabilidade e de escalonamento, alarmes e outros sinais, e tudo mais sobre o processo, as quais devem ser salvas quando o processo muda do estado em *execução* para *pronto*, a fim de que ele possa ser reiniciado posteriormente como se nunca tivesse sido interrompido.

No MINIX 3, a comunicação entre processos, o gerenciamento da memória e o gerenciamento de arquivos são tratados por módulos separados dentro do sistema, de modo que a tabela de processos é subdividida, com cada módulo mantendo os campos que precisa.

Núcleo	Gerenciamento de processos	Gerenciamento de arquivos
Registradores Contador de programa Palavra de status Ponteiro da pilha Estado do processo Prioridade de escalonamento corrente Prioridade máxima de escalonamento Tiques de escalonamento Tamanho do *quantum* Tempo de CPU usado Ponteiros da fila de mensagens Bits de sinais pendentes Bits de flag Nome do processo	Ponteiro para o segmento de texto Ponteiro para o segmento de dados Ponteiro para o segmento *bss* Status de saída Status de sinal ID do processo Processo pai Grupo do processo Tempo de CPU dos filhos UID real UID efetivo GID real GID efetivo Informações de arquivo para compartilhar texto Mapas de bits de sinais Vários bits de flag Nome do processo	Máscara UMASK Diretório raiz Diretório de trabalho Descritores de arquivo ID real UID efetivo GID real GID efetivo *tty* de controle Área de salvamento para leitura/escrita Parâmetros da chamada de sistema Bits de flag

Agora é possível explicar um pouco mais como acontecem múltiplos processos sequenciais com uma única CPU.

Associada a cada classe de dispositivo de E/S, existe uma estrutura de dados em uma tabela chamada ***tabela de descritores de interrupção***. A parte mais importante de cada entrada nesta tabela é chamada de ***vetor de interrupção***. Ele contém o endereço da rotina de tratamento do serviço de interrupção.

O contador de programa, a palavra de status do programa e, possivelmente, um ou mais registradores, são colocados na pilha pelo hardware de interrupção. Então, o

computador salta para o endereço especificado no vetor de interrupção de disco. Isso é tudo que hardware faz. Daí em diante, fica por conta do software.

A rotina do serviço de interrupção começa salvando todos os registradores na entrada da tabela de processos do processo corrente. O número do processo corrente e um ponteiro para sua entrada são mantidos em variáveis globais para que possam ser localizados rapidamente.

Então, as informações postas na pilha pela interrupção são removidas e o ponteiro da pilha é configurado para uma pilha temporária, utilizada pela rotina de tratamento de processos. Ações como salvar os registradores e configurar o ponteiro da pilha não podem nem mesmo ser expressas em linguagens de alto nível, portanto, elas são executadas por uma pequena rotina em linguagem *assembly*. Quando essa rotina termina, ela chama uma função em C para fazer o restante do trabalho para esse tipo específico de interrupção.

A comunicação entre processos no MINIX 3 ocorre por meio de mensagens, portanto, o próximo passo é construir uma mensagem para ser enviada para o processo de disco, o qual será bloqueado para esperar por ela. A mensagem informa que ocorreu uma interrupção, para distingui-la das mensagens de processos de usuário solicitando a leitura de blocos de disco e semelhantes. O estado do processo de disco agora é alterado de *bloqueado* para *pronto* e o escalonador é chamado. No MINIX 3, os processos podem ter prioridades diferentes, para fornecer um serviço melhor para as rotinas de tratamento de dispositivos E/S do que para os processos de usuário.

Se o processo for interrompido, é igualmente importante ou mais, então ele será escalonado para executar novamente e o processo de disco terá de esperar alguns instantes.

De qualquer modo, a função em C ativada pelo código de interrupção em linguagem *assembly* retorna nesse momento e esse código carrega os registradores e o mapa da memória do processo agora corrente e inicia sua execução.

O que faz o nível mais baixo do sistema operacional quando ocorre interrupção (em ordem | resumidamente)

- O hardware empilha o contador de programa
- O hardware carrega um novo contador de programa a partir do vetor de interrupção
- A rotina em linguagem *assembly* configura a nova pilha
- O serviço de interrupção em linguagem C constrói e envia a mensagem
- O código de passagem de mensagens marca como pronto o destinatário da mensagem em espera
- O escalonador decide qual processo vai ser executado em seguida
- A rotina em linguagem C retorna para o código em linguagem *assembly*
- A rotina em linguagem *assembly* inicia o novo processo corrente

1.7 Threads

Nos sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e um único fluxo de controle. Contudo, frequentemente existem situações em que é desejável ter vários fluxos de controle no mesmo espaço de endereçamento, executando quase em paralelo, como se fosse processos separados. Estes processos são chamados de ***threads***.

O outro conceito que um processo tem é o de fluxo de execução, normalmente denominado apenas por ***thread***. Uma *thread* tem um contador de programa que controla qual instrução vai ser executada. Ela possui registradores, os quais contêm suas variáveis de trabalho correntes. Possui uma pilha, que contém o histórico de execução, com um bloco para cada função chamada, mas das quais ainda não houve retorno. Embora uma *thread* deva ser executada em algum processo, a *thread* e seu processo são conceitos diferentes e podem ser tratados separadamente.

O que as *threads* acrescentam no modelo de processo é o fato de permitir que várias execuções ocorram no mesmo ambiente de processo de forma bastante independente umas das outras.

Considere um processo navegador web. Muitas páginas web contêm diversas imagens pequenas. Para cada imagem em uma página web, o navegador deve estabelecer uma conexão separada com o site de base da página e solicitar a imagem. Muito tempo é gasto no estabelecimento e na liberação de todas essas conexões. Com múltiplas *threads* dentro do navegador, várias imagens podem ser solicitadas ao mesmo tempo.

Quando múltiplas *threads* estão presentes no mesmo espaço de endereçamento, alguns dos campos não são por processos, mas por *thread*, de modo que é necessária uma tabela de segmentos separada, com uma entrada por *thread*. Entre os itens por *thread*, estão o contador de programa, os registradores e o estado.

O contador de programa é necessário porque, assim como os processos, as *threads* podem ser suspensas e retomadas. Os registradores são necessários porque quando as *threads* são suspensas, seus registradores devem ser salvos. Finalmente, assim como os processos, as *threads* podem estar no estado em *execução*, *pronto* ou *bloqueado*. A tabela lista alguns itens por processo e por *thread*

Itens por processo	**Itens por *thread***
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e rotinas de tratamento de sinal	
Informações de contabilização	

Em alguns sistemas operacionais, ele não está ciente da existência das *threads*, logo, elas são gerenciadas inteiramente em espaço de usuário. Quando uma *thread* está para ser bloqueada, ela escolhe e inicia seu sucessor, antes de parar. Vários pacotes de *threads* em

nível de usuário são de uso comum, incluindo os pacotes POSIX ***P-threads*** e Mach ***C-threads***.

Em outras, o sistema operacional está ciente da existência de múltiplas *threads* por processo, de modo que, quando uma *thread* é bloqueada, o sistema operacional escolhe a próxima a executar, seja do mesmo processo, seja de um diferente. Para fazer o escalonamento, o núcleo precisa ter uma tabela de *threads* listando todas as *threads* presentes no sistema.

Estas duas alternativas diferem consideravelmente no desempenho. A alternância entre *threads* é muito mais rápida quando o gerenciamento de *threads* é feito em espaço de usuário do que quando é necessária uma chamada de sistema.Esse fato é um agrupamento forte para se fazer o gerenciamento inteiramente em espaço de usuário. Por outro lado, quando as *threads* são gerenciadas inteiramente em espaço de usuário e uma *thread* é bloqueada, o núcleo bloqueia o processo inteiro, pois ele nem mesmo está ciente da existência de outras *threads*. Esse fato, assim como outros, é um argumento para se fazer o gerenciamento de *threads* no núcleo.

Uma classe de problemas com *threads* é o fato delas compartilharem muitas estruturas de dados. O que acontece se uma *thread* fecha um arquivo enquanto outro ainda está lendo este arquivo ? Em quase todos os sistemas que não foram projetados considerando *threads*, as bibliotecas não são reentrantes e causarão uma falha se for feita uma segunda chamada enquanto a primeira estiver ativa.

Outro problema está relacionado com o informe de erros. No UNIX, após uma chamada de sistema, o status da chamada é colocado em uma variável global *errno*. O que acontecerá se uma *thread* fizer uma chamada de sistema, e antes, que possa ler essa variável, outra *thread* fizer uma chamada de sistema, apagando o valor da original ?

Considere também os sinais. Alguns sinais são logicamente específicos a uma *thread*, enquanto outros, não. Se uma *thread* chama *alarm*, faz sentido o sinal resultante ir para a *thread* que fez a chamada. Quando o núcleo está ciente das *threads*, ele normalmente pode garantir que a *thread* correta receba sinal. Quando o núcleo não está ciente das *threads*, o pacote que as implementa deve monitorar os alarmes sozinhos.

Outros sinais, como SIGINT, iniciada pelo teclado, não são específicos de uma *thread*. Quem deve capturá-los ? Uma *thread* específica ?

Um último problema introduzido pelas *threads* é o gerenciamento da pilha. Em muitos sistemas, quando ocorre estouro de pilha, o núcleo apenas fornece mais pilha automaticamente. Quando um processo tem múltiplas *threads*, ele também deve ter múltiplas pilhas. Se o núcleo não estiver ciente de todas essas pilhas, ele não poderá aumentá-las automaticamente em caso de falta de pilha. Na verdade, ele nem mesmo percebe que uma falha de memória está relacionada com o crescimento da pilha.