1.Introdução ao escalonamento

Na época de sistemas de lote, o algoritmo de escalonamento era simples: apenas executar o próximo trabalho da fita. Em sistemas com compartilhamento de tempo, o algoritmo de escalonamento se tornou mais complexo. Em computadores pessoais, existem tarefas em segundo plano como ***daemons***. Os novos aplicativos tendem a exigir mais recursos, exigindo mais do computador.

Praticamente todos os processos alternam rajadas de computação com requisições de E/S onde a CPU executa por algum tempo sem parar e, depois, é feita uma chamada de sistema para ler ou escrever em um arquivo. Quando a chamada de sistema termina, a CPU computa novamente, até precisar de mais dados ou ter de escrever mais dados e assim por diante.

A E/S se dá quando um processo entra no estado bloqueado esperando que um dispositivo externo conclua seu trabalho. Os primeiros são chamados de processos ***limitados por processamento*** e os últimos são chamados de processos ***limitados por E/S***. Os processos limitados por processamento normalmente têm longas rajadas de uso de CPU e raramente esperam pela E/S, enquanto os processos limitados por E/S têm curtas rajadas de uso de CPU e esperas frequentes por E/S. O principal fator é o comprimento da rajada de uso de CPU e não o comprimento da rajada de E/S. É interessante notar que, à medida que as CPUs se tornam mais rápidas, os processos tendem a ficar limitados por E/S.

Existem uma variedade de situações nas quais o escalonamento pode ocorrer. Primeiramente, o escalonamento é absolutamente exigido em duas ocasiões: quando um processo termina e quando um processo é bloqueado em uma operação de E/S ou em um semáforo.

Em cada um desses casos, o processo que estava em execução se torna não apto a continuar, de modo que outro processo deva ser escolhido para executar em seguida.

Existem outras 3 ocasiões em que o escalonamento é normalmente feito, embora, não seja absolutamente necessário nesses momentos: quando um novo processo é criado; quando ocorre uma interrupção de E/S e quando ocorre uma interrupção de relógio.

Os algoritmos de escalonamento podem ser classificados em duas categorias, com relação ao modo como tratam das interrupções de relógio. Um algoritmo de ***escalonamento não-preemptivo*** seleciona um processo para executar e, em seguida, permite que ele seja executado até ser bloqueado ou até liberar a CPU voluntariamente. Em contraste, um algoritmo de ***escalonamento preemptivo*** seleciona um processo e permite que ele seja executado por algum tempo fixo máximo. Se o processo ainda estiver em execução no final do intervalo de tempo, ele será suspenso e o escalonador selecionará outro processo para executar. O ***escalonador preemptivo*** exige a ocorrência de uma interrupção de relógio no final do intervalo de tempo, para devolver o controle da CPU para o escalonador. Se não houver nenhum relógio disponível, o ***escalonamento não-preemptivo*** será a única opção.

Em diferentes ambientes são necessários diferentes algoritmos de escalonamento. Essa situação surge porque as diferentes áreas de aplicação têm objetivos diversos. É importante distinguir três ambientes: ***lote***, ***interativo*** e ***tempo real***.

Para projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Alguns destes objetivos estão listados na tabela abaixo.

Todos os sistemas	**Imparcialidade** – Dar a cada processo o mesmo tempo de uso de CPU **Imposição da política** – Garantir que a política declarada é executada **Balanceamento de carga** – Manter todas as partes do sistema ocupadas
Sistemas de lote	**Taxa de saída –** Maximizar o número de tarefas por hora **Tempo de retorno** – Minimizar o tempo entre o envio e o término **Utilização da CPU** – Manter a CPU ocupada o máximo de tempo possível
Sistemas interativos	**Tempo de resposta** – Atender rapidamente as requisições **Proporcionalidade** – Satisfazer às expectativas dos usuários
Sistemas de tempo real	**Cumprir os prazos finais** – Evitar a perda de dados **Previsibilidade** – Evitar a degradação da qualidade em sistemas mutlimídia

Sob todas as circunstâncias, a imparcialidade é importante. Processos comparáveis devem receber serviço comparável. Dar para um processo muito mais tempo de CPU do que para outro equivalente não é justo. Naturalmente, diferentes categorias de processos podem ser tratadas de formas diferentes. Pense no controle de segurança e no processamento da folha de pagamento do centro de computação de um reator nuclear.

Outro objetivo geral é manter todas as partes do sistema ocupadas, quando possível. Se a CPU e todos os dispositivos de E/S puderem ser mantidos ocupados o tempo todo, será feito, por segundo, mais trabalho do que se alguns dos componentes estiverem ociosos. Ter alguns processos vinculados à CPU e alguns processos limitados por E/S em memória é uma ideia melhor do que primeiro carregar e executar todas as tarefas vinculadas à CPU e, depois, quando elas tiverem terminado, carregar e executar todas as tarefas vinculadas à E/S.

Gerentes de centros de computação normalmente examinam três métricas para avaliarem o desempenho de seus sistemas: ***taxa de saída***, ***tempo de retorno*** e ***utilização da CPU***. A taxa de saída é o número de tarefas por segundo. Tempo de retorno é o tempo médio desde o momento em que um trabalho do lote é submetido até o momento em que ele é concluído e quanto menor, melhor.

Um algoritmo de escalonamento que maximiza a taxa de saída pode não necessariamente minimizar o tempo de retorno. A utilização da CPU também é um problema nos sistemas de lote porque, nos computadores de grande porte, onde os sistemas de lote são executados, a CPU ainda tem um custo alto. Assim, os gerentes dos centros de computação se sentem culpados quando ela não está executando o tempo todo.

O mais importante é minimizar o ***tempo de resposta***, que é o tempo entre a execução de um comando e o recebimento de seu resultado. Em um computador pessoal onde está sendo executado um processo de segundo plano, o pedido de um usuário para iniciar um

programa ou abrir um arquivo deve ter precedência sobre o trabalho de segundo plano. O fato de ter todos os pedidos interativos atendidos primeiro será percebido como um bom serviço.

Os sistemas em tempo real têm propriedades diferentes dos sistemas interativos e, assim, diferentes objetivos de escalonamento. Eles são caracterizados por terem prazos finais que devem ou pelo menos deveriam ser cumpridos. A principal necessidade em um sistema de tempo real é cumprir todos os prazos finais.

Em alguns sistemas de tempo real, especialmente aqueles que envolvem multimídia, a previsibilidade é importante. Perder um prazo final ocasional não é fatal, mas se o processo de áudio for executado muito irregularmente, a qualidade do som se deteriorará rapidamente. O vídeo também é um problema, mas os ouvidos são muito mais sensíveis à flutuação de fase do que os olhos. Para evitar esse problema, o escalonamento dos processos deve ser altamente previsível e regular.

2.Escalonamento em sistemas de lote

O algoritmo mais simples de todos de escalonamento é o ***não preemptivo FCFS*** (*First-come First-Served*). Nesse algoritmo, os processos recebem tempo de CPU na ordem em que solicitam. Basicamente, existe uma única fila de processos prontos. Quando o primeiro trabalho entra no sistema, ele é iniciado imediatamente e pode ser executado durante o tempo que quiser. Quando outros trabalhos chegam, eles são colocados no final da fila. Quando o processo que está em execução é bloqueado, o primeiro processo da fila é executado em seguida. Quando um processo bloqueado se torna pronto, assim como um trabalho recém-chegado, é colocado no final da fila.

A maior vantagem desse algoritmo é que ele é fácil de programar e também é imparcial. Com esse algoritmo, uma lista encadeada simples mantém todos os processos prontos. Selecionar um processo para executar exige apenas remover um do início da fila. Adicionar uma nova tarefa ou um processo que acaba de ser desbloqueado exige apenas inseri-lo no final da fila. Este algoritmo tem uma grande desvantagem que é depender do desempenho do computador.

Outro algoritmo ***não-preemptivo*** para sistemas de lote que presume que os tempos de execução são conhecidos antecipadamente. Quando várias tarefas igualmente importantes estão em fila de entrada esperando para serem iniciadas, o escalonador seleciona a ***SJF*** (*Shortest Job First*).

Considere os trabalhos *a*, *b*, *c* e *d*, de 8, 4, 4 e 4 minutos respectivamente. Executando-os nessa ordem, o tempo de retorno para *a* é de 8 minutos, para *b* é de 12 minutos, para *c* é de 16 minutos e para *d* é de 20 minutos, dando uma média de 14 minutos. Agora, considere a execução desses quatro trabalhos utilizando o algoritmo ***SJF***. Os tempos de retorno são de 4, 8, 12 e 20 minutos respectivamente, para uma média de 11 minutos. O algoritmo ***SJF*** provavelmente é ótimo. O primeiro trabalho acaba no tempo *a*, o segundo no tempo $a+b$ e assim por diante. O tempo de retorno médio é $(4a + 3b + 2c + d)/4$. É claro que *a* contribui mais para a média do que os outros tempos, portanto, ele deve ser o trabalho mais

curto, com *b* vindo em seguida, depois *c*, e finalmente *d*. É interessante notar que o algoritmo ***SJF*** é ótimo apenas quando todos os trabalhos estão disponíveis simultaneamente.

Uma versão preemptiva do algoritmo ***SJF*** é o algoritmo ***SRT*** (*Shortest Remaining Time Next*). Nesse algoritmo, o escalonador sempre escolhe o trabalho cujo tempo de execução restante é o mais curto. Aqui, novamente, o tempo de execução precisa ser conhecido antecipadamente. Quando chega um novo trabalho, seu tempo total é comparado com o tempo restante do processo corrente. Se um novo trabalho precisar de menos tempo para terminar do que o processo corrente, este será suspenso e o novo trabalho será iniciado. Este esquema permite que os trabalhos mais curtos recebam um bom serviço.

Quando os trabalhos chegam no sistema, eles são colocados inicialmente em uma fila de entrada armazenada em disco. O ***escalonador de admissão*** decide quais trabalhos ingressarão no sistema. Os outros são mantidos na fila de entrada até que sejam selecionados. Um algoritmo de controle de admissão típico poderia procurar uma mistura de trabalhos limitados por processamento e trabalhos limitados por E/S. Como alternativa, os trabalhos mais curtos poderiam ser admitidos rapidamente, enquanto os trabalhos mais longos teriam de esperar. O escalonador de admissão está livre para manter alguns trabalhos na fila de entrada e admitir trabalhos que cheguem depois, se optar por isso.

Quando um trabalho for admitido no sistema um processo poderá ser criado para ele e poderá competir pela CPU. O segundo nível de escalonamento é decidir quais processos teriam que ser postos no disco. Isso é chamado ***escalonador da memória***, pois eles determinam quais processos são mantidos na memória e quais são mantidos no disco. Essa decisão precisa frequentemente ser revista para permitir que os processos que estão no disco recebam algum serviço.

Se o conteúdo da memória principal é trocado com muita frequência com o do disco, isso implica que um consumo de uma grande quantidade de largura de banda de disco, diminuindo a velocidade da E/S de arquivos. Essa alternância entre estar em memória principal e estar armazenado no disco é denominado ***swapping***.

Para otimizar o desempenho do sistema como um todo, o escalonador da memória talvez queira decidir cuidadosamente quantos processos deseja ter na memória (**multiprogramação**) e quais tipos de processos. Se ele tiver informações sobre quais processos são limitados por processamento e quais são limitados por E/S, poderá tentar manter uma mistura desses tipos de processo na memória.

Para tomar suas decisões, o escalonador da memória revê periodicamente cada processo no disco para decidir se vai levá-lo para a memória ou não. Dentre os critérios que ele pode usar para tomar sua decisão são os seguintes: quanto tempo passou desde que o processo sofreu ***swap***; quanto tempo de CPU o processo recebeu recentemente; qual é o tamanho do processo; qual é a importância do processo.

O terceiro nível de escalonamento é a seleção de um dos processos prontos, armazenados na memória principal, para ser executado em seguida. Frequentemente, ele é chamado de ***escalonador da CPU*** e é o que as pessoas normalmente querem dizer quando

falam sobre escalonador. Qualquer algoritmo pode ser usado aqui, tanto preemptivo quanto não-preemptivo.

3.Escalonamento em sistemas interativos

Embora o escalonamento de três níveis não seja possível, o escalonamento de dois níveis é comum. Um dos algoritmos mais antigos, mais simples e mais amplamente utilizados é feito um ***RR*** (*Round Robin*). A cada processo é atribuído um intervalo de tempo, chamado de ***quantum***, durante o qual ele pode ser executado. Se o processo estiver em execução no fim do ***quantum***, é feita a preempção da CPU e esta é alocada a outro processo. É claro que, se o processo tiver sido bloqueado, ou terminado, antes do ***quantum*** ter expirado, a troca da CPU será feita neste momento.

O ***RR*** é fácil de implementar. Tudo o que o escalonador precisa fazer é manter uma lista de processos executáveis. O único problema interessante relacionado ao ***RR*** é a duração do ***quantum***. Trocar de um processo para outro exige certa quantidade de tempo para fazer a administração.

O escalonamento ***RR*** faz a suposição implícita de que todos os processos são igualmente importantes. Com frequência, as pessoas que possuem e operam computadores multiusuários têm ideias diferentes sobre esse assunto. Em uma universidade, a ordem da prioridade pode ser os diretores primeiro, depois os professores, secretários, inspetores e assim por diante. A necessidade de levar em conta fatores externos conduz ao ***escalonamento de prioridade***.

Para impedir que os processos de alta prioridade sejam executados indefinidamente, o escalonador pode diminuir a prioridade do processo correntemente em execução em cada tique de relógio. Se essa ação fizer com que sua prioridade caia abaixo da do próximo processo com maior prioridade, ocorrerá um chaveamento de processo. Como alternativa, cada processo pode receber um ***quantum*** de tempo máximo em que ele pode ser executado. Quando esse ***quantum*** esgota, é dada a chance de executar ao próximo processo com maior prioridade.

As prioridades podem ser atribuídas aos processos estática ou dinamicamente. O sistema UNIX tem um comando, *nice*, que permite ao usuário reduzir voluntariamente a prioridade de seu processo, para ser gentil com os outros usuários. Ninguém o utiliza.

As prioridades também podem ser atribuídas dinamicamente pelo sistema, para atender certos objetivos. Por exemplo, alguns processos são altamente limitados por E/S e gastam a maior parte do seu tempo esperando a E/S terminar. Quando um processo assim quer a CPU, deve recebê-la imediatamente para permitir que ele inicie sua próxima requisição de E/S, a qual pode então prosseguir em paralelo com outro processo que realmente faz computação. Fazer com que o processo limitado por E/S espere um longo tempo pela CPU significará apenas que ele ocupará a memória por um tempo desnecessariamente longo.

Muitas vezes é conveniente agrupar processos em classes de prioridade e utilizar escalonamento por prioridade entre as classes, mas escalonamento ***RR*** dentro de cada classe.

Um dos primeiros escalonadores por prioridade estava no CTSS, que tinha o problema de que o chaveamento de processos era muito lento. Os projetistas rapidamente perceberam que era mais eficiente dar um ***quantum*** grande para processos limitados por processamento, de vez em quando, em vez de frequentemente dar pequenos ***quantas***. Por outro lado, dar a todos os processos um ***quantum*** grande poderia significar um péssimo tempo de resposta. A solução foi configurar classes de prioridade. Os processos de classe mais alta eram executados por um ***quantum***. Os processos na classe de prioridade mais alta seguinte eram executados por dois ***quanta***.

A seguinte política foi adotada para impedir que um processo que ao ser iniciado pela primeira vez necessitasse ser executado durante um longo tempo, mas se tornasse interativo posteriormente, fosse eternamente penalizado. Quando um processo que estava esperando uma entrada de terminal era finalmente despertado, ele entrava na classe de maior prioridade. Quando um processo que estava esperando um bloco de disco tornava-se pronto, ele entrava na segunda classe. Quando um processo ainda estava em execução quando seu ***quantum*** esgotava, ele era inicialmente colocado na terceira classe.

Outro algoritmo que pode ser utilizado para fornecer resultados previsíveis de maneira semelhante, com uma implementação muito mais simples. Ele é chamado de ***escalonamento por sorteio***. A ideia básica é dar aos processos "bilhetes de loteria" para os vários recursos do sistema, como o tempo de CPU. Quando uma decisão de escalonamento tiver de ser tomada, um "bilhete de loteria" é sorteado aleatoriamente e o processo que possui esse bilhete recebe o recurso. Quando aplicado ao escalonamento da CPU, o sistema pode realizar 50 vezes por segundo, com cada vencedor recebendo como prêmio 20ms de tempo da CPU.

O escalonamento por sorteio tem algumas propriedades interessantes. Por exemplo, um novo processo aparece e recebe alguns bilhetes, no próximo sorteio ele terá uma chance de ganhar proporcional ao número de bilhetes que possui. Em outras palavras, o escalonamento por sorteio é altamente sensível.

Processos cooperativos podem trocar bilhetes se quiserem. Por exemplo, quando um processo cliente envia uma mensagem para um processo servidor e, então, é bloqueado, ele pode dar todos os seus bilhetes para o servidor, para aumentar a chance do servidor ser executado em seguida. Quando um servidor tiver terminado, ele devolverá os bilhetes para que o cliente possa ser executado novamente.

4.Escalonamento em sistemas de tempo real

Um sistema de ***tempo real*** é aquele em que o tempo desempenha um papel fundamental. Normalmente, um ou mais dispositivos físicos externos ao computador geram estímulos e o computador deve interagir apropriadamente a eles, dentro de um período de tempo fixo.

Os sistemas de tempo real geralmente são classificados como de ***tempo real rígido***, significando que há prazos finais absolutos a serem cumpridos, e de ***tempo real relaxado ou não-rígido***, significando que perder um prazo final ocasionalmente é indesejável, mas tolerável. Em ambos os casos, o comportamento de tempo real é obtido dividindo-se o programa em vários processos, cujo comportamento é previsível e conhecido

antecipadamente. Geralmente, esses processos têm vida curta e podem ser executados até o fim em menos de um segundo.

Os eventos a que um sistema de tempo real responde podem ser classificados mais especificamente como ***periódicos*** ou ***aperiódicos***. Um sistema pode ter de responder a vários fluxos de eventos periódicos. Dependendo de quanto tempo cada evento exigir para processamento, talvez nem seja possível tratar de todos eles.

5.Política Vs Mecanismo

Admitimos tacitamente que todos os processos no sistema pertencem a usuários diferentes e assim estão competindo pela CPU. Embora isso frequentemente seja verdadeiro, às vezes acontece de um processo ter muitos filhos executando sob seu controle. Por exemplo, um processo de sistema de gerenciamento de banco de dados pode criar muitos filhos. É plenamente possível que o processo principal tenha uma excelente noção de quais de seus filhos são os mais importantes e quais são os menos importantes.

A solução para esse problema é separar o ***mecanismo de escalonamento*** da ***política de escalonamento***. Isso significa que o algoritmo de escalonamento é parametrizado de alguma maneira, mas os parâmetros podem ser fornecidos pelos processos de usuário.

Suponha que o núcleo utilize um algoritmo de escalonamento por prioridade, mas forneça uma chamada de sistema por meio da qual um processo pode configurar e alterar as prioridades de seus filhos. Assim, o pai pode controlar com detalhes como seus filhos são postos em execução, mesmo que não faça o escalonamento em si.

6.Escalonamento de threads

Quando vários processos têm múltiplas *threads* cada um, temos dois níveis de paralelismo presentes: processos e *threads*. O escalonamento em tais sistemas difere substancialmente, dependendo se as *threads* são suportadas em nível de usuário ou em nível de núcleo.

Como o núcleo não tem conhecimento da existência de *threads*, ele funciona normalmente, selecionando um processo e dando a esse processo o controle do seu ***quantum***. Dentro do processo existe um escalonador de *threads* que seleciona qual *thread* vai executar. Como não existem interrupções de relógio para multiprogramar as *threads*, essa *thread* pode continuar executando o quanto quiser.

Quando o processo finalmente for executado outra vez, a *thread* retornará a sua execução. Ela continuará a consumir todo o tempo do processo até que termine. Entretanto, seu comportamento anti-social não afetará outros processos. Eles receberão o que o escalonador considerar como sua fatia apropriada, independente do que estiver acontecendo dentro do processo.

Uma diferença importante entre *threads* em nível de usuário e *threads* em nível de núcleo é o desempenho. Fazer um chaveamento de *threads* em nível de usuário exige algumas instruções de máquina. As *threads* em nível de núcleo exigem uma troca de contexto

completa, alterando o mapa de memória e invalidando o cache, o que é muitas vezes mais lento. Por outro lado, com *threads* em nível de núcleo, bloquear uma *thread* em E/S não suspende o processo inteiro, como acontece com *threads* em nível de usuário.

Outro fator importante a considerar é que as *threads* em nível de usuário podem empregar um escalonamento específico do aplicativo. Por exemplo, considere um servidor web que possui uma *thread* "despachante" para aceitar e distribuir os pedidos recebidos para *threads* operárias. Suponha que uma *thread* operária tenha acabado de ser bloqueada e que a *thread* despachante e duas *threads* operárias estejam prontas. Quem deve ser executado em seguida? O escalonador, sabendo o que todas as *threads* fazem, pode selecionar facilmente a despachante, para permitir que se possa pôr outra operária em execução. Essa estratégia maximiza o volume de paralelismo em um ambiente onde as operárias são frequentemente bloqueadas em E/S de disco. Com *threads* em nível de núcleo, o núcleo nunca saberia o que cada *thread* faz. Em geral, contudo, os escalonadores de *threads* específicos do aplicativo podem otimizar sua execução melhor do que o núcleo.