Tópicos

introdução sistemas operacionais, process abstraction, dual-mode operation, mode transfer, segurança, memória, interrupções

Podemos entender o sistema operacional como um gerenciador de recursos que cria e mantém processos isolados (***A***). Esses processos têm algumas liberdades para acessar e escrever dados, e realizar pedidos, dentro de seus invólucros, através de (***B***), implementadas pelo sistema operacional de forma segura através de ***C, D***, entre muitas outras.

(1) system calls, privileged instructions

(2) timer interrupts

(3) process isolation

(4) mode transfers

Associe corretamente as opções acima com as letras no enunciado e descreva brevemente o que é e como funciona cada uma das opções, utilizando os termos técnicos pertinentes.

Resposta

A–3 B–1 C–4 D–2

1: ***system calls:*** ocorrem quando um processo rodando em modo usuário faz uma chamada ao kernel, disponibilizada e definida pelo sistema operacional. Essa chamada causa a mudança de modo de operação do processador de usuário para kernel. Em outras palavras, entrega o processador para uma parte definida do kernel, que então faz o seu trabalho (I/O, por exemplo) utilizando ***privileged instructions.*** Ao terminar, entrega o processador para o processo que o chamou, restaurando as condições anteriores.

2: ***timer interrupts:*** são interrupções implementadas em hardware, podendo ocorrer por tempo ou por número de instruções executadas, por exemplo. Essas interrupções garantem que o processador seja entregue ao kernel constantemente, a fim de que o sistema operacional possa conferir se está tudo sob controle (encerrar loops infinitos, por exemplo).

3: ***process isolation:*** é a criação de processos em um ambiente cujos recursos são controlados pelo sistema operacional, de forma que a segurança e estabilidade do sistema não dependa do processo (dos programas). É implementado utilizando-se os conceitos deste exercício.

4: ***mode transfers:*** é a transferência de modo de operação do processador entre usuário (menos privilegiado) e kernel (mais privilegiado). Ocorre em ***system calls*** (processos), ***time interrupts*** (hardware), ***external interrupts*** (dispositivos) e ***processor exceptions***. Ao transferir o modo de operação, o estado do processador (registradores, flags) e do processo (stack de memória) são salvos na memória (com auxílio do hardware e do software). Após o tratamento da ocorrência (***handler***), o estado anterior é restaurado. Para melhorar a segurança e estabilidade, adota-se ***two-stacks*** de memória por processo (user e kernel), o que facilita mudar processos dentro de uma interrupção, além de ***interrupt masking***, uma espécie de scheduler de interrupções, para organizar interrupções de interrupções.

Tópicos

gerenciamento de processos, shell, processo pai e filho, signal, entrada e saída, dispositivos

Problema

1. Nos primeiros sistemas de processamento, usuários submetiam trabalhos e o sistema operacional assumia todo o controle de enfileirar e instanciar esses trabalhos para execução. Como é feito o gerenciamento de processos (trabalhos) nos sistemas atuais? Dê um exemplo de como funciona para um sistema conhecido.
2. Sistemas computacionais possuem uma infinidade de dispositivos, como teclados, mouses, placas, câmeras. Esses dispositivos são feitos por muitos fabricantes diferentes e possuem muitas funcionalidades. Qual foi a solução desenvolvida para o sistema operacional lidar com tantos equipamentos, com o mínimo de adaptação constante necessária?

Resposta

3. Sistemas modernos permitem que programas, inclusive do usuário, criem e gerenciem seus próprios processos. Exemplos são gerenciadores de janelas, navegadores de internet, interpretadores de comandos de ***shell***. Aqui cabe uma explicação para o ***shell***. ***Shell*** é um programa que contém uma sequência de tarefas, cada uma podendo ser um outro programa. Assim, pode ser um programa de arquivos de programas. Pode enviar suas saídas para um arquivo ou receber entradas de um arquivo, ou pode alimentar um as entradas de outro programa. Um exemplo de funcionamento pode ser dado pelo UNIX. São necessárias 2 chamadas ao sistema para criar um processo: ***fork*** e ***exec***. Fork cria uma cópia do ***processo pai*** (aquele que chamou o ***fork***), um novo ***processo filho*** com algumas modificações que o diferenciam do processo pai, de preparo para execução do programa que está para ser executado, como privilégios, I/O, abertura ou fechamento de arquivos, entre

outros. Como ele roda o mesmo código do pai, pode ser confiado para preparar o ambiente para o novo programa. Terminado o ***fork***, o ***exec*** é chamado. ***Exec*** carrega na memória o executável, efetivamente iniciando o seu funcionamento. Uma das funcionalidades disponíveis é o ***signal***, uma notificação que pode ser enviada de um processo para outro, utilizado para finalizar, suspender, ou resumir processos, por exemplo.

4. Sistema operacionais utilizam uma interface para padronizar a interação de dispositivos com o kernel. No UNIX, a entrada e saída de dispositivos e arquivos, além da comunicação interprocessos, foi padronizada em uma única interface comum, através de 4 chamadas ao sistema (***open, close, read, write***), uma solução utilizada até hoje e adotada universalmente. Antes de ler ou escrever, o processo pede para abrir (o dispositivo, arquivo, etc). Isso permite ao sistema operacional checar permissões, e realizar tarefas pertinentes internamente. Toda comunicação é feita em arrays de bytes, mesmo que de tamanhos diferentes. ***Kernel-buffered*** reads e writes faz com que a transmissão de dados seja canalizada em um buffer no kernel, o que permite padronizar o tratamento de diferentes tipos de transmissão (streaming, blocos), além de permitir a execução de outras tarefas enquanto aguarda por dados, ou a execução do programa enquanto espera o dispositivo responder à escrita desse mesmo programa (quando o programa gera dados mais rápido do que podem ser recebidos pelo dispositivo). Quando o processo explicita um close, o sistema operacional pode realizar rotinas pertinentes, como coletar todos os dados não mais necessários e jogá-los fora. ***Pipes***, outro conceito importante, é um buffer de duas vias (leitura e escrita) e desacopla a relação de espera de uma escrita-leitura, aumentando a velocidade. ***Replace file descriptor*** permite que o processo filho não precise saber de onde está vindo o I/O.

Tópicos

concurrency e threads, thread abstraction, thread management (independent, shared data, life cycle, state), kernel threads, multi-threaded processes, kernel context-switch

Problema

Explique o conceito de simultaneidade (concurrency) e threads. Por que são importantes, quais são suas vantagens em um sistema computacional? Como o kernel gerencia threads?

Resposta

Simultaneidade (***concurrency***) refere-se a múltiplas atividades que podem ocorrer ao mesmo tempo. A vida é o maior exemplo disso, onde muitas coisas estão ocorrendo

simultaneamente. Em sistemas computacionais, threads permitem que o computador execute múltiplas tarefas simultaneamente e paralelamente, ao invés de uma única tarefa sequencialmente, trazendo grandes benefícios para o funcionamento de processos. Por exemplo, em um aplicativo de mapas, podemos dividir o processamento em threads para receber comandos do usuário, carregar partes da interface gráfica e requisitar dados a um servidor, tudo simultaneamente. Caso contrário, poderíamos ter de esperar um servidor responder para carregar o menu da aplicação. Explicando de forma simples, um programa sequencial pode ter sua execução dividida em partes, onde cada parte é executada sequencialmente e simultaneamente com outras partes. Essas partes podem ser independentes, ou podem se comunicar com outras partes. Como o programador não tem total controle de quais threads serão executadas em qual período de tempo (pois muitas threads são executadas e pausadas devido à diferentes níveis de prioridades, além de que o processador pode ser interrompido ou ter sua velocidade (clock) alterada por diversos motivos, como eficiência energética), threads que compartilham estruturas de dados precisam ser sincronizadas. Threads são criadas e gerenciados de forma semelhante a processos, e possuem funções ***create, yield*** [executar ou pausar, definido pelo ***scheduler***], ***join*** [thread aguarda um evento ou chamada de outra thread] e ***exit***. O kernel decide qual thread será executada ou será pausada através de um ***scheduler***, assim como define em qual processador físico ela será executada, definindo dessa forma sua ***life-cycle***. Assim, o número de threads não está vinculado ao número de processadores reais. O sistema operacional cria uma ilusão de que um programa pode criar e utilizar quantas threads quiser. Threads podem ser independentes, ou podem ser dependentes de outras threads. Assim, cada thread (***per-thread***) possui seu ***thread control block*** (TCB) onde são armazenadas informações independentes sobre cada thread (thread ID, prioridade de scheduling, dono, registradores e pointer para o stack), e também tem um ***shared state*** (código do programa, variáveis estáticas globais e a heap), um espaço para compartilhamento de estruturas de dados entre threads. Kernels modernos também utilizam threads para realizar mais trabalho em em menos tempo. Threads podem funcionar ao fundo do sistema, realizando tarefas paralelamente a outras tarefas. A mudança de threads (***scheduling***) sendo executadas recebe o nome de ***thread context-switch.*** Pode ser voluntária, quando uma thread chama outra thread (de uma biblioteca por exemplo), ou involuntária, quando ocorre uma interrupção ou exceção no processador. Interrupções podem ser desabilitadas durante thread switches para evitar confusão. Threads também são utilizadas para melhorar a latência de dispositivos de ***input/output***, podendo servir como buffers de dados. Um processo pode ser executado em várias threads (***multi-threaded process***). Assim, computadores executam múltiplos processos, onde cada processo é executado em múltiplas threads simultaneamente, alocados em múltiplos processadores (cores ou CPUs), cujos recursos são gerenciados e alocados pelo kernel.

Tópicos

thread synchronization, race condition, locks, condition variables, shared objects, bounded queues, semaphores

Problema

Explique o conceito e a importância da sincronização de threads. Quando ocorre uma condição de corrida? O que são objetos compartilhados?

Resposta

Sincronização de threads (***structured cooperating threads synchronization***) é um método sistemático para gerenciar a comunicação entre threads que lêem e escrevem dados compartilhados. Threads podem ser independentes (executam como um programa sequencial único independente), ou podem depender e se comunicar com outros threads. Essa comunicação cria um problema, como a ***race condition***. Caso um thread faça operações com variáveis que dependem ou fazem parte de outro thread, o resultado dependerá de qual thread for executado primeiro, por exemplo. Compiladores alteram a ordem de execução das instruções (***out-of-order***), processadores possuem velocidade variável, assim como executam processos diferentes a cada momento, a depender de ***interrupts*** e ***schedulers***, por exemplo. Assim, existem inúmeras possibilidades de combinação da ordem de execução, o que impossibilitaria a programação para utilizar compartilhamento eficiente entre threads. A solução para essa questão foi a adoção de uma interface ***orientada a objetos***, de forma que em todo momento apenas um thread possui o acesso (***lock***) a um objeto compartilhado (***shared objects***), podendo realizar modificações que, para outros threads que utilizam o mesmo objeto, parecem ser ***atômicas***. Ou seja, outro thread não precisa saber como o dado compartilhado foi alterado. Assim que um thread libera um objeto compartilhado, outro thread assume o controle, e assim sucessivamente. ***Shared objects*** são armazenados em ***heap*** por questão de segurança, e possuem ***synchronization variables***, uma estrutura de dados utilizada para coordenar acessos ao ***shared state***. São duas: ***locks*** (utilizado para exclusão mútua de threads, ou seja, apenas um thread realiza modificações ao objeto compartilhado por vez) e ***condition variables,*** utilizadas para um thread aguardar outro thread realizar uma ação em uma ***thread-safe bounded queues*** (filas de tamanho limitado e seguras para utilização por threads para inserir e retirar dados), para que um thread não trabalhe em filas vazias, por exemplo. Em threads, ***critical section*** é o nome dado para a parte do código que acessa um objeto compartilhado. Métodos de sincronização também desabilitam interrupções para tornar as modificações em shared state atômicas. Um método de sincronização alternativo ao que utiliza ***locks*** é o ***semaphore***, bastante utilizado para gerenciar interrupções de I/O do hardware. Nesse caso, um algoritmo com uma variável decimal de controle é utilizado para exclusão mútua e espera entre threads.

Tópicos

multi-object atomicity, acquire-all/release-all, serializability, two phase locking, deadlock, nested waiting, starvation, preventing deadlocks

Problema

Explique conceitos e a importância de métodos de acesso a múltiplos shared objects. O que é deadlock? O que é starvation? Como evitar um deadlock?

Resposta

Na medida em que um programa com múltiplos threads possui múltiplos shared objects, é necessário avaliar como os acessos a esses objetos é feito, pois uma sequência de acessos atômicos não é uma operação atômica. Uma forma para lidar com essa questão é o **acquire-all/release-all,** que consiste em adquirir todos os locks necessários de uma vez. Quando possui todos os locks, a thread executa as operações, e libera os locks. Essa técnica permite significativo paralelismo para pedidos individuais de dados não compartilhados, e possui a propriedade de serializabilidade, onde pensa-se que cada pedido é executado em ordem um por um. Porém, para isso, é preciso saber quais locks serão necessários. Nesse sentido, temos o **two phase locking**: locks podem ser feitos na medida em que são necessários, mas não são liberados sem que todos os locks para a operação sejam conseguidos. São posteriormente todos liberados no final do pedido. Assim não é necessário saber quais locks serão necessários em antemão. **Deadlock** é o termo utilizado para a ocorrência um ciclo de espera entre um conjunto de threads, onde cada thread espera outro fazer alguma coisa. Pode ocorrer em muitas situações, como em **mutually recursive locking**: ocorre quando um thread possui um lock em um objeto, e outro thread possui lock em um segundo objeto, e cada thread faz um pedido para o outro objeto mantendo seu atual lock. Dessa forma, os dois threads vão aguardar infinitamente pois nenhum libera o lock para o outro continuar. Outra situação é **nested waiting**, quando um shared object chama outro shared object enquanto mantém o lock no primeiro objeto, e aguarda por uma variável de condição. O wait_for_signal libera o lock no segundo objeto, mas não no primeiro. Se o objeto geraria o sinal precisa do lock no primeiro objeto, ele nunca consegue sinalizar, e ocorre deadlock. Em resumo, deadlock pode ocorrer em qualquer situação onde um thread espera um evento que nunca vai acontecer. Também podem ocorrer por falta de recursos ou falta de espaço, como memória, tempo de processamento, entrada e saída e buffers. **Starvation** é quando um thread não consegue progredir por um tempo indefinido de tempo. Deadlock é uma forma de starvation, porém com um grupo de threads que não consegue progredir em um ciclo de espera entre eles. Um deadlock implica em starvation, mas starvation não implica em deadlock. Um sistema está propenso a starvation se um thread pode sofrer starvation a qualquer momento sob algumas circunstância. O mesmo se aplica a deadlock. Fatores como os mencionados acima, além das escolhas feitas pelo **scheduler**, o número de threads simultâneos, a carga de trabalho e muitos outros, são circunstâncias que podem causar starvations e deadlocks. Podemos citar 4 condições para a ocorrência de deadlock: número finito de recursos que threads podem utilizar, a propriedade de um lock só pode ser mudada pelo proprietário, um thread aguarda enquanto espera por um recurso, e espera circular, onde um thread está esperando o outro thread que está esperando o outro thread, em um ciclo infinito de espera. Para evitar deadlock, pode-se: limitar o comportamento do programa para evitar essas situações, predizer o futuro ao saber o que threads vão ou podem fazer, detectar e recuperar de deadlocks, prover recursos suficientes, permitir que o sistema proclame a propriedade sobre um lock para tentar destravar o ciclo de espera, liberar locks quando precisar aguardar um recurso em um módulo externo e a capacidade do sistema organizar a ordem dos locks e apenas seguir aquela ordem correta.

Tópicos

address translation (benefícios e implementações), virtual and physical addresses, segmented memory, paged memory, multi-level translation, paged segmentation, cache, translation lookaside buffer, virtually and physically addressed caches, cache access methods, cache update policies, cache replacement policies

Problema

Explique o que é address translation. Quais são seus benefícios? Fale sobre as diferentes implementações e tipos de organização da memória. Por que utilizar cache? Como caches funcionam?

Resposta

Address translation é a conversão de endereços virtuais de memória em endereços da memória física, que cria uma abstração (ilusão), da perspectiva dos programas e programadores, de que a memória é contínua e ilimitada. Por trás disso, o sistema operacional utiliza técnicas para gerenciar a memória de forma segura, confiável e rápida para ele próprio e os processos. ***Address translation*** permite o isolamento dos processos, a comunicação entre processos, o compartilhamento de código e dados, a inicialização antecipada de programas, possibilita o gerenciamento dinâmico e eficiente da memória, a implementação de ***caches***, o controle de I/O, program debugging, entre outros. ***Virtual addresses*** são os endereços que o processo enxerga, e não correspondem, necessariamente, à realidade física da memória. Na memória física, estão os ***physical addresses***, os locais reais da memória. O mecanismo de tradução é o responsável por fazer a ponte entre o virtual e o real, de forma a garantir todos os benefícios mencionados anteriormente. Esse mecanismo pode ser feito com software e hardware. Uma das formas mais simples de designar espaços de memória é através de ***bases*** e ***bounds***, que definem um começo e um fim de espaço de memória permitido para um processo. Essa informação é mantida por registradores no processador. Para resolver a limitação pelos registradores, que precisam ser salvos e mudados em todo context switch, existe o conceito de ***segmented memory***: um array de pares de bases e bounds para cada processo (***segmentation***), então cada par controla um segmento contínuo de espaço de memória em endereços virtuais, mas os segmentos podem estar fisicamente espalhados pela memória. Com segmentos, o sistema operacional pode permitir que alguns segmentos sejam compartilhados por processos, assim como processos podem compartilhar código utilizando diferentes dados. Bibliotecas também podem ser compartilhadas por processos, enquanto os dados para cada processo ficam separados. Também permite comunicação entre processos, caso seja permitido leitura e escrita para o mesmo segmento. Também permite gerenciamento eficaz da memória para reutilização, zerando o conteúdo de segmentos rapidamente. Um problema é para memórias alocadas dinamicamente, onde não se sabe quanta memória um programa poderá consumir. Isso pode ser melhorado com ***zero-on-reference***. Outro problema é que, a

medida que a memória vai ficando segmentada, pode ocorrer de ter espaço disponível, porém não contínuo. Isso se chama ***external fragmentation***, e então o sistema operacional precisa compactar a memória, que tem um custo. Um método alternativo é ***paged memory***: a memória é alocada em pedaços (páginas) de tamanho fixo, chamadas de page frames. O address translation é semelhante ao da memória segmentada, porém cada processo contém ponteiros para as páginas. Esse sistema resolve o principal problema da segmentação: alocação de espaço livre. Agora, o sistema operacional vê a memória física como um mapa de bits, onde cada bit representa uma página que está ocupada ou vazia. Encontrar um espaço livre é só encontrar um bit livre. Todas os benefícios mencionados são atingidos através dessa forma de organização. Uma nova vantagem é a inicialização antecipada. Um programa pode começar a ser executado antes de que todas as páginas necessárias sejam carregadas do disco para a memória. Se ocorrer da execução precisar de uma página ainda não carregada, ocorre uma exceção que aguarda o carregamento. O compilador também pode reordenar as instruções para facilitar esse mecanismo. Também permite ***data breakpoint***, uma forma de parar a execução de um programa quando este acessa uma localização de memória. Um problema de memórias por páginas é que o gerenciamento da memória virtual é mais complexo, e o tamanho da tabela de páginas é proporcional ao tamanho do espaço de endereçamento virtual, não à memória física. Qual o tamanho certo para uma página? Uma página muito grande desperdiça espaço caso o processo não a utilize completamente. Outro ponto é como procurar os endereços para conversão de virtual para físico. Muitos sistemas se baseiam em árvores de busca para a tabela de páginas (***multi-level translation***). Outro conceito é ***paged segmentation***: a memória é segmentada, mas ao invés de cada segmento apontar para um espaço contínuo de memória, cada segmento aponta para uma página, que aponta para o pedaço de memória guardando aquele segmento. Uma forma de tornar a conversão de endereços mais rápida, e portanto o acesso à memória mais eficiente, é utilizar ***cache***. Cache é uma memória que contém uma pequena quantidade de dados que podem ser acessados muito rápido. ***Caches*** possuem aplicações abrangentes, mas vamos falar do seu uso para ***address translation***. Assim, pode-se falar do ***translation lookaside buffer (TLB)***, uma tabela em hardware que contém os resultados de conversões recentes de endereços. Ou seja, uma cache de address translations. Cada entrada na tabela representa o mapeamento de uma página virtual em uma página física. Se ocorre um hit, o processador já sabe imediatamente o endereço que precisa. Caso contrário, ocorre a conversão normalmente, que leva mais tempo (a tabela de páginas é acessada da memória, o endereço é procurado, e então entregue para o processador). Quando ocorre context switch, TLBs podem passar por ***flush***, onde são limpas para receber os endereços de outro processo. ***Address aliasing*** é quando mais de um endereço aponta para o mesmo espaço. ***Caches*** funcionam com princípio de proximidade local e temporal (***temporal*** e ***spatial locality***), ou seja, existe uma alta chance de um dado próximo ser usado, e de ser usado mais de uma vez. São memórias pequenas, porém muito rápidas, que guardam dados que, pelos princípios descritos, serão bastante utilizados e úteis, tornando o carregamento de dados mais rápido. Assim, são utilizadas em muitas aplicações, como com conteúdo de internet, buscas, memória virtual, sistema de arquivos. Cache ***hit*** é quando o dado está na cache e pode ser utilizado imediatamente, e ***miss*** quando não está, e precisa ser carregado da memória. Os endereços podem ser acessados na cache de diversas formas, como ***direct mapped***: cada endereço só pode ser guardado em um local definido na tabela, o que torna a procura pelo endereço rápida (literalmente, se o endereço bate é um hit, senão miss), porém muito inflexível. ***Fully associative***: cada endereço pode ser salvo em qualquer lugar, e o sistema precisa checar endereço por endereço para encontrar o que procura (mais lento), porém é muito flexível para descartar endereços. Por fim, ***set-associative***: uma mescla de direct mapped e fully associative, um pouco mais lenta que direct mapped, e um pouco menos flexível que fully associative. Para ***caches write-through*** todas as suas mudanças (na cache)

são igualmente realizadas na memória. Para caches ***write-back***, só atualizam a memória quando a cache está sem espaço. ***Memory hierarchy*** é o conceito de representar os níveis de memória (cache, DRAM, disco) por atributos de velocidade e tamanho. Existem diferentes políticas de troca endereços da cache: ***aleatória, FIFO, LRU, LFU***).

Tópicos

processor scheduling policy, task, response time, predictability, throughput, scheduling overhead, fairness, starvation, workload, work-conserving and preemptive, time quantum, multi-level feedback queue (MFQ)

Problema

O que é scheduling? Quais são os termos importantes? Quais são as políticas mais gerais?

Resposta

Em sistemas computacionais é comum existir mais tarefas do que recursos para as executar. ***Scheduling*** é a técnica utilizada para decidir quais tarefas executar e em qual ordem, e a resposta para essa questão é bastante dependente do tipo de tarefa, portanto não existe uma única resposta correta. Para cada solução existe um trade-off de propriedades que melhoram ou comprometem o desempenho. Nesse campo, é importante explicar alguns termos que serão utilizados posteriormente. ***Task***, ou ***job***, pode ser de qualquer tamanho e tipo, e não equivale a uma thread ou processo necessariamente. Por exemplo, cada letra digitada em um processador de texto é uma task. ***Response time***, ou ***delay***, é o tempo percebido pelo usuário para a execução de uma task. ***Predictability*** é a baixa variância entre response times para tasks repetidas. ***Throughput*** é a taxa em que tasks são concluídas. ***Scheduling overhead*** é o tempo gasto para trocar a execução de uma task para outra. ***Fairness*** é a igualdade dada em tempo de execução e recursos para cada task. ***Starvation*** é quando uma task não progride, devido aos recursos estarem sendo dados para outras tasks de maior prioridade, por exemplo. ***Workload*** é um conjunto de tasks para serem concluídas juntamente com o momento de início e quanto tempo leva para completar. Assim, dado um ***workload***, o scheduler define quando cada task é escolhida para ser executada. A execução de tasks depende do tipo de task. Existem as que são inteiramente computacionais (***compute-bound***), outras inteiramente de entrada e saída (***I/O-bound***), assim como misturadas. As políticas de scheduling dependem, portanto, dos tipos de tasks. Considera-se, ainda, que o scheduling é ***work-conserving*** (o processador nunca fica livre se há trabalho a fazer) e ***preemptive*** (o scheduler pode tirar uma task do processador e colocar outra quando quiser). A política ***FIFO*** executa uma task até que ela termine, diminuindo overhead. Se o número de tasks é fixo e são todas compute-bound, FIFO têm o melhor throughput, além de ser a própria definição de fairness (cada task espera a sua vez, que chegará). Entretanto, se com frequência uma task de execução rápida acaba atrás de uma de execução longa, o sistema não é eficiente. A política ***SJF*** (***shortest job first***) é a política que minimiza o average response time, pois as

tasks que possuem a menor quantidade de trabalho para ser realizada são executadas primeiro. A quantidade de trabalho é sempre contada em tempo real (diminui com a execução). Entretanto, a sua implementação requer conhecer o trabalho de cada task antecipadamente. Além disso, tarefas longas são fortemente penalizadas e demoram muito para executar, e o sistema pode sofrer starvation devido às constantes trocas de contexto, fazendo com que tarefas longas possam nunca completar se sempre existir tarefas mais curtas. Outra política é o ***Round Robin***, que lida com starvation. Nessa forma, todas as tasks possuem um tempo definido para ficar executando, definido por um **time *quantum***. Definir esse intervalo é importante, pois se for muito curto ocorre muita troca de task, o que impacta a performance. Essa política também não funciona bem para tasks de entrada e saída, pois estas geralmente ocupam pequenos períodos do processador para fazer o pedido de I/O. ***Max-Min Fairness*** diz que a alocação suficiente de recursos é tão importante quanto a política do scheduler para melhorar a responsividade e diminuir o ***overhead***. Essa técnica interativamente maximiza a alocação mínima de recursos dados para um processo particular, até que todos os recursos estejam sendo utilizados. Se todas as tarefas são ***compute-bond***, utiliza portanto ***Round Robin***, que é o mais indicado nesse caso. ***Multi-level feedback queue*** (***MFQ***) é a utilização de diversas políticas simultaneamente, tentando aproveitar os benefícios de cada uma de acordo com a necessidade do momento, e é a técnica utilizada comercialmente pelos sistemas operacionais, e tenta conquistar um compromisso para as situações mais comuns do mundo real. Assim, essa política utiliza múltiplas filas ***Round Robin***, cada uma com diferentes níveis de prioridade e time quantum. Tasks com maior prioridade causam preempt de tarefas de menor prioridade, e tarefas de mesma prioridade são escolhidas com a política ***Round Robin***. Maior prioridade possui menor ***time quantum*** do que menor prioridade. Tasks movem entre níveis de prioridade para favorecer tasks menores (mais rápidas). Novas tasks entram no topo do nível de prioridade. Toda vez que uma task utiliza seu time quantum, ela cai de prioridade. Toda vez que uma task entrega o processador aguardando I/O, seu nível de prioridade é mantido. Essa metodologia mantêm os recursos alocados sempre em utilização, e os distribui de maneira inteligente para melhorar a responsividade e diminuir o ***response time***.

Tópicos

file system abstraction, non-volatile storage, persistent data, named data, file, metadata, directory, path, hard link, volume, file descriptor, application libraries, buffering, block cache, prefetching, device drivers, device access, direct memory access, magnetic disk, platters, arm, head, access optimization, flash storage, flash translation layer, wear

Problema

O que é a abstração do sistema de arquivos? Quais são seus termos mais importantes? Como funciona o endereçamento e a interface com os dispositivos? Explique brevemente sobre discos magnéticos e armazenamento flash.

Resposta

Sistemas de arquivos são uma abstração do sistema operacional que permite aplicações acessar informações não voláteis (***persistent data***) e ***named data*** (identificador associado com o arquivo, permitindo acesso e compartilhamento independente de um programa específico), armazenados em um ***non-volatile storage***. Tecnologias de armazenamento do tipo citado possuem suas próprias limitações, como é o caso da diferença de performance de um disco rígido e de chips de memória principal (DRAM). Essas limitações são melhoradas através de técnicas no sistema de arquivos, por exemplo. Um sistema de arquivos deve ser confiável, possuir alta capacidade de armazenamento e ter baixo custo, alta performance, e permitir o compartilhamento controlado de dados. Um ***file*** é uma coleção de dados em um sistema de arquivos, permitindo que uma quantidade quase arbitrária de dados seja representada por um nome. Um ***file metadata*** são as informações sobre o ***file***, utilizada e gerenciada pelo sistema operacional, como tamanho, tempo de modificação e dono. O ***file data*** são quaisquer dados contidos no file, escritos por algum programa. ***Directory*** são nomes que mapeiam o sistema de arquivo em uma abstração de pastas. ***Path*** é o caminho de um diretório, e pode ser relativo ao atual diretório ou ao diretório raiz do sistema. ***Volume*** é uma coleção de recursos de armazenamento que formam um dispositivo de armazenamento lógico. Criar, abrir, editar, e deletar files podem ser feitos através de chamadas, como open() e read(). O acesso pode ser feito através de ***file descriptor***, pois a permissão do file, por exemplo, só pode ser feito após a abertura, e não precisa ser repetida em todas as chamadas àquele arquivo. Quando um processo abre um arquivo, o sistema operacional cria uma estrutura de dados que armazena as informações sobre os arquivos abertos pelo processo. O ***file descriptor*** é como uma referência do sistema operacional, arquivo por arquivo, utilizado pelo sistema operacional para gerenciar o acesso ao arquivo pelo processo. ***Application libraries*** também podem englobar as chamadas ao sistema para acesso a arquivos para adicionar outras funcionalidades, como ***buffering***. Outras técnicas que melhoram a performance são ***block cache*** (armazena dados recentes), e ***prefetching*** (acessa blocos próximos antecipadamente). Device drivers traduzem abstrações de alto nível implementadas pelo sistema operacional para as particularidades do hardware dos dispositivos de armazenamento. ***Block device*** é um bloco de tamanho definido em que os dados são lidos ou escritos. O acesso a dispositivos de entrada e saída (***device access)*** pode ser feito através de ***memory-mapped I/O***. Assim, os registradores do dispositivos são mapeados como endereços de memória, e controlados dessa maneira. ***Direct memory access*** ocorre quando o dispositivo pode copiar o que está na sua memória diretamente na memória principal. A notificação de que o dispositivo terminou de ler/escrever um dado pode ser realizado através de interrupção. ***Magnetic disks*** são dispositivos mecânicos de armazenamento amplamente utilizados em computadores, constituídos de discos (***platters***) que rodam de 4200 a 15.000 RPM, e braços com cabeça de leitura (***arm's head***) (que detectam ou inserem campos magnéticos no disco). Bits de dados são armazenados em setores (***sectors***) de tamanho fixo (512 bytes por exemplo), e são lidos setores inteiros. Um círculo de setores no disco é chamado de trilha (***track***). Os dados em uma ***track*** podem, portanto, serem lidos ou escritos sem a movimentação do braço. Esses dispositivos também possuem um ***buffer memory***, utilizado para os dados sendo escritos ou lidos. ***Track buffering*** armazenam as trilhas que foram lidas pelo disco mas não requisitadas pelo sistema operacional, para melhorar a performance. Os setores são identificados como ***logical block addresses***, que especificam qual surface, track e sector deve ser acessado. Para melhorar a performance, a ordem do acesso de dados também é definida (***FIFO***, ***SPTF*** e ***SSTF*** são possibilidades), para reduzir o movimento do braço (alto custo de tempo de acesso). ***Flash storages*** não possuem partes móveis, os dados são armazenados com circuitos elétricos

(portas lógicas), e assim são muito mais velozes que discos magnéticos, e possuem ***flash translation layer*** que mapeia páginas flash lógicas em diferentes páginas flash físicas. Sua durabilidade também é menor, podendo sofrer ***wear*** (células de armazenamento deixam de funcionar).

Tópicos

file systems, directories, index structure, free space maps, locality heuristics, defragmentation, fat, ffs, ntfs

Problema

Como funciona um sistema de arquivos de maneira geral? Quais são os sistemas de arquivos mais utilizados?

Resposta

A implementação de um sistema de arquivos trata-se de resolver um problema de tradução de nomes de arquivos em números e blocos. Sistemas de arquivos podem implementar uma espécie de dicionário que mapeia nomes de arquivos em números de blocos em um dispositivo. A maioria das implementações são baseadas em 4 idéias: diretórios, estruturas de índices, mapas de espaço livre e heurísticas de localidade. Utilizam ***directories*** para mapear nomes (legíveis por humanos) para números de arquivos (são como arquivos especiais que contém listas de mapeamento de nomes para números), e depois utilizam uma ***index structure*** (como uma árvore) para encontrar o bloco que guarda o arquivo. Também é implementado o ***free space map*** para mapear quais blocos estão livres e quais estão sendo utilizados conforme arquivos crescem ou diminuem. Dessa forma, sistemas de arquivos podem encontrar arquivos e seus metadata não importa onde estejam guardados, assim como os espaços livres e ocupados. Dessa maneira, muitas políticas são adotadas para decidir onde guardar esses arquivos, baseadas em ***locality heuristics*** para agrupar dados e melhorar a performance, como por exemplo reunir os arquivos de mesmo directory, e directories espalhados em diferentes partes. Outra técnica é a ***defragmentation***, que reorganiza a distribuição dos arquivos no disco em sequência por exemplo. Sistemas de arquivos como o ***FFS*** (Unix e derivados como ext2 e ext3 no Linux) e ***NTFS*** (evolução do ***FAT*** no Windows, cujas técnicas também aparecem no Linux ext4, Journaled File System e no Apple HFS) utilizam árvores de múltiplos níveis para a index structure. NTFS utiliza uma árvore ainda mais flexível, assim como otimiza sua index structure para arquivos sequenciais. A estrutura de dados em árvore melhora o acesso aleatório, assim como utilizam coleções de locality heuristics para obter uma melhor localidade espacial. FAT é considerado o mais simples, e utiliza uma lista ligada, e é amplamente utilizado em pen drives e armazenamentos mais simples, devido a sua simplicidade e interoperabilidade entre diversos sistemas. Esse sistema de arquivos possui baixa performance para acesso aleatório, possui metadata e controle de acesso limitados, e não possui suporte para hard links, assim como sua tabela de

índices impõem um limite no tamanho de armazenamento suportado (1TB para blocos de 4KB).