Introdução a Cadeira

Como cada máquina tem um conjunto diferente de instruções para comandá-las (ou seja, são heterogêneas, que leva a sistemas complexos), então é necessário criar um camada que abstraia essa complexidade para os programadores de aplicativos. Este camada é a Infraestrutura de Software.

A infraestrutura de Software são definidas como os sistemas operacionais e o middleware, que gerenciam a memória de modo a facilitar a usabilidade do computador, de forma menos complexa tanto para o desenvolvedor de aplicativos finais quanto para o próprio usuário, que não precisa ter acesso direto a máquina.

Sistema Operacional

O sistema operacional é dito como um programa que tem acesso total e direto ao hardware, podendo executar qualquer operação que a máquina seja capaz. Além disso, o SO impede que o usuário faça modificações prejudiciais ao hardware.

As duas funções claras que o SO executa o distingue em dois aspectos: visto como um programa de abstração sobre o hardware e visto como um gerenciador de recursos do mesmo.

- Visão de máquina estendida
  Operar sobre a arquitetura da máquina as vezes (para não dizer muitas) é uma tarefa enfadonha. Por essa razão, é necessário abstrair o certos conceitos para a aplicação de modo que ela só saiba o essencial. Esta é a tarefa do SO, criar abstrações, implementá-las e gerenciar os objetos abstratos, sendo estas abstrações claras, precisas, elegantes e coerentes. Exemplos dessas abstrações podem ser a própria interface gráfica do Windows ou o prompt de comando; ambos são modos diferentes de interagir com o hardware.
  Podemos afirmar que esta é uma visão top-down (abstração do hardware para interfaces (partes) mais convenientes)
- Visão de gerenciador de recursos
  Esta é já uma visão bottom-up, onde o SO fica responsável por gerenciar todo o hardware (alocação de memória, controle de dados, fluxo, etc). É deste modo que conflitos como acesso à dados são impedidos de acontecer, além de prioridades e repartição de tempo.
  Podemos entender isso melhor observando a CPU: vários programas precisam utilizar a CPU, fica por responsabilidade do SO administrar quem usa quando. Além disso, o SO também tem de administrar quanto espaço ficará alocada para cada aplicação de modo a otimizar o utilização do hardware
  Esse compartilhamento de recursos para muitos programas é chamado de multiplexação.

História dos sistemas operacionais

Vamos começar com um fato importante, a primeira programadora do mundo e contratada por Babagge para construir um sistema operacional para sua máquina analítica, Ada Lovelace Byron.

Vamos começar a história agora:

1. Primeira Geração (1945-1955) – Válvulas

   Sua principal característica é usar válvulas para realizar processamentos, por esse motivo são consideradas primitivas e lentas (levavam segundos para fazer cálculos simples). Funções eram feitas pela conexão de plugs em painéis. A linguagem era de máquina pura e não havia sistema operacional.

   Para executar um programa, o procedimento era bem incomum: reservava-se um tempo de máquina para o programa e ia a sala de máquinas executá-lo e esperar que desse certo (seja no tempo seja não queimando as válvulas).

2. Segunda Geração (1955-1965) – Transistores e lotes (batch)

   Avanço da tecnologia e tempo de vida dos computadores, criando os computadores de grande porte (*mainframes*). O processo era bem parecido com o anterior, no entanto já havia uma linguagem de programação (Assembly,Fortran) e usava-se cartões perfuráveis para entregar o programa à sala de entradas e a saída era impressa na sala de saída. Processo ainda lento pela falta de compilação.

   Para agilizar este processo, foi criado o sistema em lotes, onde toda a entrada era guardada em um fita, onde o computador processava etapa por etapa da fita e a gravava em outra fita através de um programa especial (predecessor do SO). A fita de saída era levada para outra máquina que imprimia as respostas dadas *off-line*, ou seja, fora da máquina principal.

   Os cartões de controles foram os precursores das linguagens de controle de tarefas e dos interpretadores do comandos atuais.

3. Terceira Geração (1965-1980) – Cis e Multiprogramação

   Surgimento dos Circuitos Integrados (CI). Necessidade de criar um software que pudesse ser implementado em qualquer modelo de máquina da mesma "família", inclusive o sistema operacional.

   Ocioso grande parte do tempo, visto que apenas uma ação podia ser executada por vez, a memória foi repartida em partes de modo que cada tarefa teria seu espaço e a CPU e os componentes de entrada ficariam livres para processar aqueles que os convocavam.

   Com a necessidade de melhor compartilhamento de tempo – o sistema de lotes não estava sendo benéfico para a maioria dos programadores – foi criado a ideia de "tempo compartilhado", onde cada usuário tinha acesos a um pedaço de tempo da CPU, aproveitando horas vagas ao máximo.

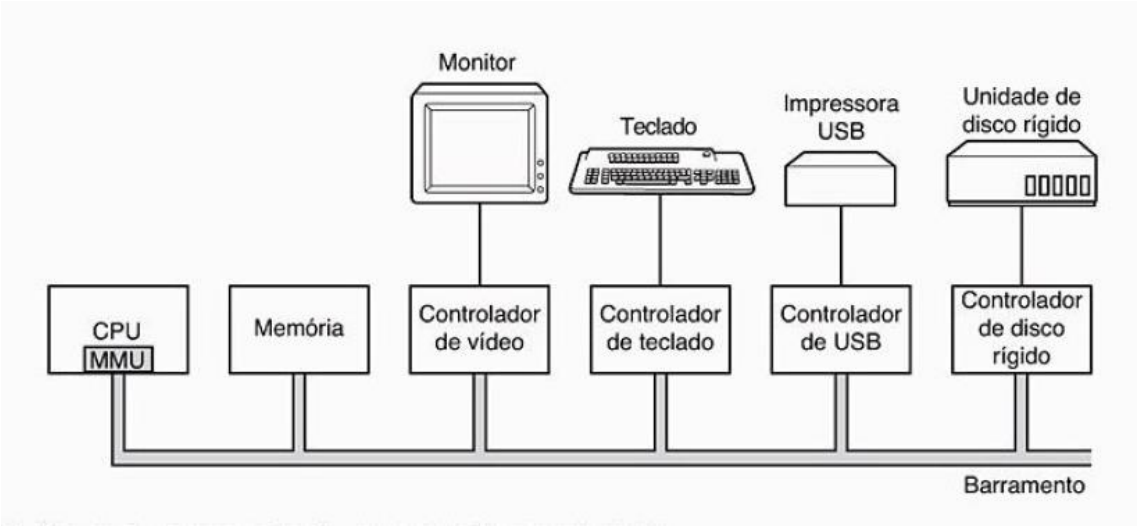
   Além disso, surgiu a necessidade de criar um sistema operacional que servisse tanto para grandes processamentos (científicos) quando para os mais simples (menos complexos).

   Com a permissão de multiprogramação, SO podia ficar operando o tempo todo enquanto outros programas rodavam.

4. Quarta Geração (1980-atuais) – Computadores Pessoais

Revisão sobre hardware

Temos de ter essa breve revisão pois o sistema operacional está intimamente ligado ao hardware, ele tem de ter total entendimento e controle das partes físicas do computador. A figura abaixo ilustra um computador simples.



Processadores

Podemos resumir os Processadores a CPU, o centro de operações do computador. A CPU segue um ciclo: buscar operação, decodificar operação, executar operação. Além disso, é importante ressaltar que CPUs tem instruções específicas, ou seja, um programa para a CPU A não pode ser executada pela CPU B e vice-versa.

CPUs tem a característica de terem registradores internos para facilitar o acesso aos dados e agilizar o processo de execução da tarefa. Por esse motivo, é normal ter instruções que mechem com os registradores, além das instruções aritméticas e lógicas.

Concomitantemente, na CPU há o contador de programa (*Program Counter – PC*) que define em que registrador está a próxima instrução a ser executada. Desse modo, a CPU sempre sabe onde está a instrução que ela precisa executar.

Outros registradores na CPU que são fundamentais são: PSW (*program status word*), responsável pela chamada de sistemas e em entradas e saídas, e o ponteiro de pilha, que aponta para o primeiro item na pilha da memória (pilha de rotinas que estão em processamento – imaginar como se fosse literalmente uma pilha de recurssão).

Uma função importante da CPU é manter os estados dos registradores de cada programa que está sendo executado. Quando o sistema operacional dá uma parcela de tempo para um programa, este programa executa o que puder e ao fim, salva seus estados nos registradores que a CPU mantém controle, de modo que, quando novamente o programa receber uma fatia de tempo, ele poder voltar ao estado que estava.

Memória

A memória deveria ser a parte do sistema mais rápida de acessar, barata e grande para suportar muitos dados. No entanto, isto não é verdade e varia de

tecnologia para tecnologia, por isso foi criado a hierarquia de memória, onde as camadas superiores tem maior velocidade, menor capacidade de armazenamento e são mais custosas/bit do que as camadas inferiores. Vamos analisar:

1. Registradores → Feitos do mesmo material da CPU e não fornecem atraso a ela, no entanto seu tamanho é de 32x32 bits ou 64x64 bits (baixíssimo).
2. Memória Cache → Mantidas próximas ou dentro da CPU, servem como um acesso rápido a dados muito utilizados. Além disso, nos computadores modernos, há dois caches, L1 e L2, um dentro da CPU e com pouca capacidade de armazenamento, mas alta velocidade de acesso, enquanto que o outro fica fora e armazena mais palavras de instruções, mas é mais devagar.
3. Memoria Principal → É onde a CPU procura suas requisições quando elas não se encontram na cache. Existem memórias adjuntas a esta que não são voláteis, como a ROM – responsável pela inicialização do computador.
4. Disco Rígido → Memória secundária de acesso física com grande capacidade de armazenamento, mas baixa velocidade.

Estrutura de Sistemas Operacionais

Vamos analisar as principais características de 4 estruturas de sistemas operacionais de modo a entender como eles funcionam internamente:

Sistemas Monolíticos

- O sistema operacional é executado como um único programa no modo núcleo (com acesso a todas as operações de hardware) – um processo com n procedimentos.
- Formado por um conjunto de rotinas que podem chamar quaisquer outras, tornando o SO mais complexo.
- Tem a seguinte organização hierárquica:
    1. Um programa principal que invoca a rotina a ser requisitada;
    2. Um conjunto de rotinas de serviço que executam as chamadas de sistema (*system call*) – ou seja, para cada chamada do sistema, há uma rotina de serviço que se encarrega dela;
    3. Um conjunto de rotinas utilitárias que auxiliam as rotinas de serviço.

Sistemas de Camadas

- O sistema é totalmente hierárquico, onde cada camada possui sua função bem definida.
- Abstração da função de cada camada, onde a camada superior não necessitava saber de dados da camada inferior, apenas recebe-los.
- Esse tipo de estruturação permite criar um sistema operacional flexível (facilita evolução e adaptação a novos ambientes) e modular (separada em módulos).

Sistemas de modelo Cliente-Servidor

- Ideia básica de cliente faz uma requisição ao servidos enviando os dados necessários, o servidor processa o pedido e retorna com uma resposta.
- As aplicações são consideradas clientes enquanto que aqueles que executam os processos são considerados os servidores (servidor de serviço).
- Em um sistema operacional distribuído (ou seja, em várias máquinas), os serviços voltados para as aplicações ficam distribuídos em diferentes máquinas. A única camada em comum entre essas máquinas é o núcleo, que permite a comunicação.
- Abstração da localidade da máquina – não importa se este modelo é feito localmente ou entre máquinas diferentes via rede, a ideia é a mesma para ambos os casos.
- É um sistema que permite processamento rápido da informações em um servidor de grande capacidade.

Infraestrutura para SDs: SOD, SOR e Middleware para modelo Cliente-Servidor

- SOD – Sistemas operacionais Distribuídos
  São sistemas fortemente acoplados para sistemas e programas distribuídos, uma visão única e global dos recursos. Nesta arquitetura, o Kernel é igual para todas as máquinas e o sistema operacional e as aplicações ficam distribuídas. Esta é uma arquitetura ótima para máquinas coma:
    - Sistemas homogêneos
    - Transparências de distribuição (abstração do SO)
    - Alto desempenho
    - Memória compartilhada
    - Controle de concorrência
- SOR – Sistemas Operacionais de Rede
  São sistemas fracamente acoplados onde cada máquina execute seu próprio SO e um SO completo para cada máquina. Temos aqui além do Kernel repetido, o sistema operacional de cada um é único e apenas as aplicações são distribuídas. Ótimo para máquinas com características que combinam melhor com a realidade:
    - Sistemas heterogêneos
    - Pouca transparência
    - Escalabilidade
    - Comunicação

  No entanto, para auxiliar nessa abstração dos SORs, é necessário um middleware que complementa as abstrações. Apesar de ainda ter o sistema operacional único, a existência desta camada de middleware facilita bastante o trabalho do SO. Uma máquina ideal para tal arquitetura é:
    - Sistemas heterogêneos
    - Transparência de comunicação e distribuição

- Serviços e Abertura (independe do SO para comunicação)

Sistemas de virtualização

- Criação do monitor de máquina virtual – uma cópia do hardware é feita para cada máquina virtual criada, podendo assim cada uma ter seu próprio controle sobre o hardware e seu próprio sistema operacional.
- Separação das funções de multiprogramação e máquina estendida de modo a ter partes mais flexíveis, simples e de fácil manutenção.
- É uma camada acima do hardware, mas abaixo do sistema operacional, onde cada aplicação é executada de acordo com o sistema operacional em que a aplicação se encontra.
- É um tecnologia antiga, desde os anos 60, que permitia ter acesso à conjunto diferente de instruções de acordo com a aplicação.
- Temos as seguintes arquiteturas para esse tipo de sistema:
  - Hipervisor Tipo 1
    Há uma mesma camada para todos os sistema operacionais para eles se comunicarem com o hardware com uma única camada de abstração.
    Roda em modo Kernel (núcleo)
  - Hipervisor Tipo 2
    Existe ainda um sistema operacional principal e existe uma camada acima dele que permite que outros SOs existam na máquina, no entanto, seus programas passam pelo Hipervisor Tipo 2, depois pelo *Host* e então seguem para o hardware, gerando um processo mais lento. Podemos dizer que este tipo é na verdade uma aplicação do sistema operacional principal.

Mais um pouco de Processos

Conceito de Processo

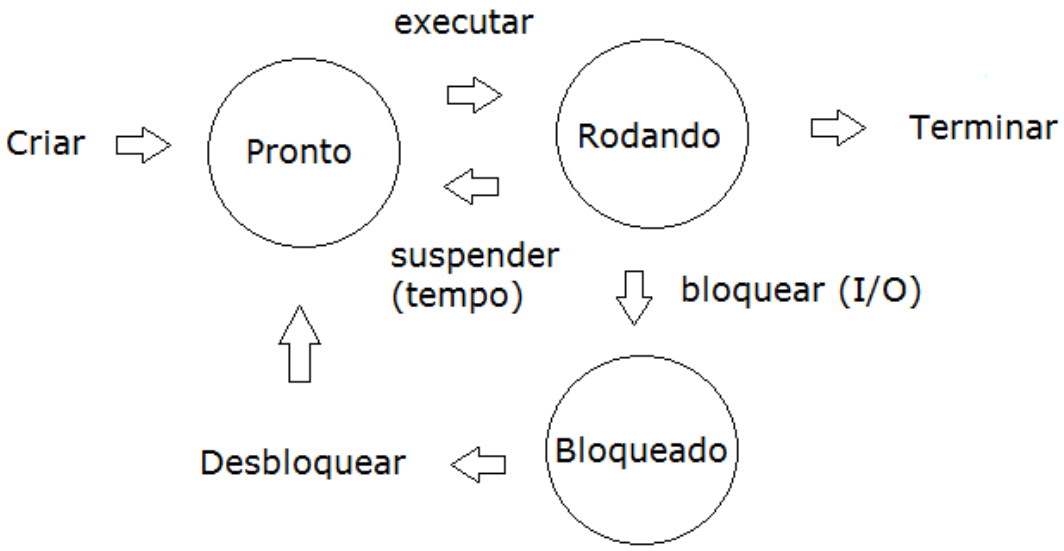
Um processo é um programa em execução. Ou seja, após um programa ser compilado e virar um programa executável e carregado na memória, suas instruções podem então ser lidas na memória, tornando assim um processo.

Para um processo, temos o seguinte contexto (estado):

- CPU: Registradores
- Memória: Posições em uso
- E/S: Estado das requisições
- Estado do Processo: Rodando, bloqueado, Pronto
- Outras (que são a depender do processo)

Temos isso porque há necessidade de obter requisições, pelo processo, para que ele execute, logo é necessário ter conhecimento delas e saber se estas requisições estão disponíveis.

Máquina de Estado do Processo



- Pronto
  Como se pode perceber, um processo pode entrar neste estado em três situações diferentes e estará pronto para ser executado pela CPU a qualquer momento:
  - Criado: O processo foi criado por alguém e está esperando o escalonador o chamar;
  - Suspenso: O tempo de execução do processo expirou e este deve dar a vez para outro processo;
  - Desbloqueado: Um recurso que o processo esperava foi lhe dado e este está pronto para ser chamado
- Rodando
  A CPU está executando / rodando o processo chamado pelo escalonador. Caso o processo termine no tempo estipulador para ele, este sairá da memória.
- Bloqueado
  Quando um processo estava rodando na CPU mas necessitou de um recurso que ainda não se encontrava disponível para ele, este deve ser bloqueado e esperar pelo recurso.

Ciclo de Vida de Um Processo

Para cada um dos ciclos de vida de um processo, há uma correlação do processo com a máquina como um todo. O ciclo é formado por três principais ramos criação, execução e término. Vejamos, por cima, como um processo pode ser criado ou terminado:

I. Criação de Processo:
   Principais eventos que levam a criação de processo:
   - Início de Sistema,

- Execução de chamada ao sistema de criação de processos,
- Solicitação do usuário para criar novo processo,
- Início de um job (processo) em sistema em lote.

Quando um sistema é inicializado, ele gera processos que ficam no *background* da máquina, sem interação com o usuário. Esses processos são conhecidos por daemons.

II. Término de Processos:

Condições que levam ao término de Processo:

- Saída Normal (voluntária),
- Saída por erro (voluntária),
- Erro fatal (involuntário),
- Cancelamento por outro processo.

Hierarquia de Processos

É a possibilidade de que um processo pode gerar novos processos. De modo semelhante, estes novos processos pode gerar novos processos, com conceito de processo "pai" e seus "filhos" (INUX chama isso de grupo de processos). No caso do Windows, apesar de existir a Hierarquia, a relação de pai e filho não existe entre processos, são criados com níveis iguais.

Implementação de Processos

Para implementar o modelo de processo, o SO mantém uma tabela de dados sobre este processo de modo a registrar todo o seu andamento e, caso este seja bloqueado ou ido para o estado de pronto, a sua tabela de dados será salva de modo que, ao voltar para a execução, parecerá que o processo nunca saiu. Esta tabela é chamada de tabela de processos ou blocos de controle de processos.

Multiprogramação

É a ideia de ter múltiplos programas em execução, se tornando possível com a grande capacidade de armazenamento da memória e aumento da velocidade da CPU em um processo, podendo suportar mais de um processo carregada em si e uma grande velocidade de processamento.

Apesar dos programas, aparentemente estarem sendo executados em paralelo, são executadas em estilo de compartilhamento de tempo, onde cada um recebe uma fatia de tempo para executar alguma instrução, baseado no escalonamento, que diz qual é o próximo processo a ganhar uma fatia de tempo. Quando o processo é interrompido, todo o seu contexto é guardado na memória para, quando este for novamente executado, pode saber qual foi seu último estado.

Além disso, existe um fator chamado de grau de multiprogramação, que é definido como a probabilidade da CPU estar ocupada com algum processo. Este cálculo é definido como 1 – tempo de espera por entrada e saída. É através deste cálculo que se sabe até quando aumentar a memória de modo a torná-la mais eficiente e menos custosa.

Escalonamento de Processos

Quando dois ou mais processos estiverem pronto para serem executados, o sistema operacional tem de decidir qual deles vai ser executado primeiro. O responsável pela decisão é o escalonador, que possui um algoritmo chamado de escalonamento (existem vários e são bastante velhos).

Para que um processo não execute tempo demais (como antigamente era feito), os computadores modernos possuem um mecanismo de relógio chamado *clock* que causa uma interrupção periódica do processo.

Interrupção

Para controlar entrada e saída de dados, não é interessante que a CPU tenha de ficar continuamente monitorando o status dos dispositivos como discos ou teclados. Mecanismos de interrupção são criados para chamar a atenção da CPU quando algo deve ser feito.

Existem três (meio que dois) tipos de interrupção:

- Interrupção assíncrona ou de Hardware:
  Ocorre independente das instruções que a CPU está executando e gerada por dispositivos externos a CPU.

  Quando ocorre uma interrupção, a CPU interrompe o processamento do programa em execução e executa uma rotina do SO chamado de tratador de interrupção, que trata a interrupção, e posteriormente, volta o controle para o programa anterior.

  Além disso, não há qualquer comunicação entre o programa interrompido e o tratador.
- Interrupção de relógio (um tipo especial de Assíncrona):
  Quando o tempo para um processo excede seu limite de tempo (*quantum*), o relógio avisa a CPU para interromper sua execução e passar a ação para o escalonador para decidir quem é o próximo processo.
- Interrupção Síncronas ou *Traps* ou por software:

  Ocorrem em consequência da instrução sendo executada (no programa em execução), ou seja, há uma relação entre a interrupção e o programa em execução, sendo o processo responsável por solicitar a interrupção.

  Pode ser gerada pelo hardware nas situações em que o programa não teria como prosseguir (memória limite, etc.).

  Neste caso há uma comunicação entre o tratador e o programa que pediu interrupção.

Podemos então analisar o seguinte fato: dado que um processo rodando pede uma interrupção e vai para um instrumento externo a CPU, esta interrupção é classificada como *trap*. Quando o instrumento externo terminar o processo recebido, este interrompe o SO para alertar que acabou, sendo agora a interrupção assíncrona, ao mesmo tempo, caso haja algum processo cujo *quantum* tenha terminado, o relógio alerta o SO para interrompê-lo, e temos então uma interrupção de relógio.

Vale ressaltar que um programa não pode chamar diretamente uma rotina do sistema operacional, já que o SO é um processo a parte, o que ele faz é ativar um tratador que pode então enviar a chamada ao Sistema Operacional.

Vamos analisar agora o passo-a-passo que acontece:

1. O hardware primeiramente detecta que ocorreu uma interrupção,
2. Aguarda o final da instrução da execução corrente e aciona o tratador,
3. Salva o contexto de execução do processo interrompido, como PC e os status dos registradores, sendo salva pelo tratador de interrupção numa lista independente para as interrupções.

Chamadas aos Sistema

Podemos dizer, inicialmente, que uma chamada ao sistema é o uso efetivo da habilidade do SO de abstrair a complexidade das operações, de modo que o programa faz uma chamada ao sistema aparentemente simples para o usuário, e o SO realiza a parte complexa desta chamada.

Existem os seguintes tipos de chamadas ao sistema:

- Gerenciamento de Processos;
- Gerenciamento de Arquivos;
- Gerenciamento de Diretório;
- Diversos (tempo, diretório de trabalho, proteção de arquivo, sinal ao processo).

Processos e Threads

Nós já estudamos o que são os processos e como ocorre mais de um processo ser executado em pseudo-paralelismo. Vamos agora entender melhor os algoritmos de escalonamentos, aqueles que são capaz de dizer que processo irá ser executado naquele clock, e vamos aprender o que é Thread e como elas são utilizadas.

Algoritmos de Escalonamento

Já sabemos que são esses os algoritmos responsáveis por alocar os processos na memória, de modo a efetivamente utilizar a máquina. Mas é necessário que ele obedece alguns ideais:

- Ideal de Justiça
  Todo processo que pedir para ser executado, deve ser executado.
- Ideal de Eficiência
  Tem-se de dar tempo suficiente para um processo de modo que ele execute efetivamente, não necessariamente completamente.
- Ideal de Preempção
  Um processo é dito de preemptivo caso este seja capaz de ser interrompido a qualquer momento durante execução, dando a chance para outros processos.

Mas nenhum algoritmo consegue fazer esses três ideais concomitantemente, logo ele irá falhar em pelo menos um. Um algoritmo de escalonamento ótimo é aquele que consegue chegar perto do ideal.

Threads

Normalmente, para cada linha de execução (*thread*), temos um processo associado. No entanto, isso acaba impedindo o melhor uso da memória para a execução das linhas e também do ideal de paralelismo, pois cada processo tem de esperar sua fez para ser executado, concorrendo entre si a nível de núcleo (dando tempo mais para processos do que para *threads*). Concomitantemente, o espaço de endereçamento de um não é acessível para o outro, impedindo eles de se ajudarem.

Threads foram criadas para que, em um único processo, várias linhas de execução pudessem ser executadas paralelamente, uma comunicando com a outro, no mesmo e único processo, ou seja, a concorrência pudesse existir. Apesar de serem "paralelas", as *threads* continuam brigando por recursos, só que no nível de usuário. Vale ressaltar que existe um estilo de sistema onde o gerenciamento de *threads* é feito à nível de núcleo, onde este já supões que *threads* existem, diferentemente do anterior – não perde tempo descobrindo que existem *threads* para serem iniciadas e a nem todos *threads* de um processo necessariamente será executado na vez do processo.

Para não deixar dúvidas, ainda existe um escalonamento a nível de usuário com relação as *threads*, no entanto, o tempo que o SO dará a um processo será repartido entre suas *threads* através de um meta-escalonamento.

Uma *thread* também é conhecida como um processo leve, visto que uma *thread* apenas existe devido a um processo, logo seus itens de controle são bem menos, em quantidade, e também são privados, ou seja, são unicamente visto pelo processo que o chamou, diferentemente do processo, que tem variáveis compartilhadas para que outros processos e o próprio SO seja capaz de visualizar.

Outra característica das *threads*, semelhantes aos processos, são seus ciclos de vida. Possuem o mesmo ciclo de estado de um processo, podendo ser em execução, bloqueada ou pronta. Existe também uma *thread* especial, chamada de *thread* pop-up. Esta *thread* ela é criada no momento de chegada de uma mensagem / dados e esta nova *thread* é responsável por processá-los. O interessante dela é que ela não possui qualquer histórico, sendo sua criação quase imediata.

Mas como nem tudo são flores, existem problemas para a concorrência das threads:

- Não-determinismo
  Duas *threads* executando, não se sabe qual *thread* irá primeiro ser executada.
- Dependência de velocidade
  Duas *threads* que realizam uma função, não se sabe qual função terminará primeiro.
- Starvation
- Deadlock
  Duas *threads* dependem concomitantemente uma da outra e não podem avançar por causa que o recurso de uma está a depender da outra.

Tipos de Processos

Existem dois tipos de processos, que basicamente se diferenciam em que local ele passa mais tempo:

- CPU-bound / orientada a computação
  Quando o processo passa a maior parte do seu tempo na CPU. Processos com longos tempos de execução e baixo volume de comunicação entre processos.
- I/O-bound
  Quando o processos passa a maior parte do seu tempo esperando por dispositivos de entrada e saída.

Processos I/O-bound devem ter prioridade sobre processos CPU-bound, pois os processos I/O-bound "dão" sua vez para os processos abaixo dele ao irem para o dispositivo de entrada e saída, aumenta então a vazão da CPU (quantidade de processos executada na CPU por uma intervalo de tempo).

Vantagens da Implementação de Thread

Podemos dizer que sua principal vantagem é simplificação; transformar um processo inteiro em processos menos, concorrentes, ágeis e com um modelo de programação mais simples.

Estas são as mesmas vantagens dos processos, se refletirmos um pouco. No entanto, sua grande diferença é a capacidade de compartilhamento do mesmo espaço de endereçamento e todos os seus dados entre elas, o que gera uma gama de aplicações. Além disso, se fossemos criar / deletar um processo em vez de uma *thread*, isto nos levaria muito mais tempo.

A seguinte tabela comprova o porquê das *threads* serem utilizadas tanto:

Modelo	**Característica**	**Vantagens**
Threads	Paralelismo, chamada de sistemas bloqueantes	Programação simples Alto desempenho
Monothreads	Não paralelismo, chamadas de sistemas bloqueantes	Programação simples Baixo desempenho
Máquina de estado finitos	Paralelismo, chamadas de sistemas não-bloqueantes, interupções	Programação complexa Baixo desempenho

Deixemos mais claros e melhor explicado: caso o sistema seja CPU-bound (utilizam a CPU), não há ganho significativo de desempenho. No entanto, na realidade dos tempos de hoje, com grande fluxo de dados para os dispositivos de entrada e saída, existe um aumento de desempenho, pois, dado que uma *thread* irá se executada em I/O, então a CPU ficará esperando pela *thread* seguinte.

Além disso, como atualmente, é comum os sistemas terem multicores, criando de fato um paralelismo real e *thread* serem altamente recomendada.

Implementação no espaço de usuário X núcleo

Como já vimos anteriormente, podemos implementar um pacote de *threads* de dois modos diferentes, no espaço do usuário e no espaço do núcleo. Iremos agora analisar a profundo estas implementações e verificar suas vantagens e desvantagens.

No espaço do usuário, o núcleo fica livre do gerenciamento das *threads*, apenas responsável pelo gerenciamento dos processos (monothreads), deixando para a camada do usuário o gerenciamento das *threads*. A mais óbvia vantagem disso é que, independente de como o SO reage a *threads*, tal implementação será funcional. Ou seja, caso o SO não seja compatível com *threads*, como estas não estão implementadas no núcleo dele, processos podem possuir *threads* em que eles mesmo gerenciam. Para isso, é necessário que o processo mantenha o controle das *threads*, sendo isto feito através de uma tabela de *threads*, análoga a tabela de processos.

Um forte argumento para a implementação no espaço de usuário é que desviar o controle para o núcleo requer tempo, quando uma rotina de tratamento local de *threads* poderia existir para gerenciar a tabela de *threads* rapidamente e outra rotina para escalonar as *threads* locais, agilizando a execução e sem necessidade de passar o comando para o núcleo.

Apesar disso, com a implementação em usuário, não se sabe como fazer para que *threads* que foram bloqueadas não afetam todas as outras *threads* do mesmo processo. Existem alguns soluções, no entanto, estas se tornam inviáveis, como alterar o modo como os sistemas operacionais funcionam alguns funções ou reescrever partes da biblioteca de chamadas ao sistema, onde um *jacket / wrapper* (jaqueta) verifica se a chamada ao sistema causará bloqueio do *thread*. Além disso existe o problema com "falta de páginas", ou seja, quando uma *threads* pede parte do processo que ainda não foi enviado para a memória e nisto o SO tem de carregar esta nova página, bloqueando o processo como um todo (visto que ele não sabe o que é uma *thread*). Entretanto, apensar de todos estas desvantagens, a principal de todas é uma pergunta: por que implementar um sistema de *threads* se o processo não se interrompe nunca?

Vamos agora estudar a implementação em espaço de núcleo. Diferentemente da de usuário, não existirá tabela de *threads* em cada processo, existe uma única tabela de *threads* no núcleo, e também não existe um rotina para escalonamento de *threads* em cada processo. Ou seja, todo o gerenciamento de *threads* fica por responsabilidade do núcleo e a chamadas de sistemas. Uma característica peculiar nisso é que o núcleo pode decidir se, quando uma *thread* de um processo A é bloqueada, a próxima *thread* a ser executada é a do mesmo processo ou de um processo diferente.

Além disso, não há necessidade de haver chamadas ao sistemas não-bloqueante, visto que não importa se uma *thread* foi bloqueada ou não, e o defeito de falta de página da implementação de usuário é corrigido aqui, pois o núcleo pode dar a vez para a *thread* do mesmo processo para esta utilizar o tempo em que a *thread* inicial foi bloqueada para a busca de página.

Apesar disso, como já foi dito anteriormente, o custo temporal de uma chamada ao sistema é consideravelmente maior que a de uma rotina do próprio processo, sendo este o seu maior gargalo. Para minimizar este tempo foi criada a ideia de reciclagem de *thread*. Uma vez que uma *thread* é deletada, esta não perde sua estrutura de dado no núcleo, apenas não será mais executada. Então, quando uma nova *thread* for criada, esta usará o "material" da antiga *thread* para ser desenvolvida, poupando tempo de inicialização.

Como ambas possuem vantagens muito boas, há projetos que tentar uni-las, de forma a gerar um sistema de gerencia algumas *threads* no núcleo e pode multiplexar

essas *threads* em *threads* de usuário, economizando tempo e tendo um melhor escalonamento das *threads*.

Existe um método que ele busca aperfeiçoar a implementação de usuário de modo a torná-la menos problemática e mais vantajoso como a implementação de núcleo, ao mesmo tempo que continua rápida. Este é o método de ativação de escalonador. Ele usa a ideia de evitar chamar o núcleo a menor quantidade de vezes possível, sem impedir que haja chamadas ao sistema bloqueantes ou necessidade de verificar se uma chamada irá bloquear a *thread*. Além disso, existe o que se chamada de processadores virtual, semelhantes a CPU, onde cada processo ganha uma certa quantidade de processadores virtuais para alocar seus *threads*.

O método, em sua essencial, é simplificado da seguinte maneira:

1. Quando uma *thread* é bloqueada por qualquer razão, o núcleo ficarás sabendo e emite uma alerta (*upcall*) para a rotina de escalonamento de *threads* do processo (sistema de tempo de execução).
2. Ao perceber o alerta, o sistema de tempo de execução tem a permissão de reescalonar suas *threads*, colocando outra *thread* em execução e mudando o valor do estado da antiga para bloqueada.
3. Quando a *thread* estiver novamente pronta, outro *upcall* é emitido pelo núcleo para informar o sistema de tempo de execução.
4. Com o novo alerta emitido, o sistema poderá colocar a antiga *thread* na lista de prontos, indicando que poderá executá-la.

No entanto, existe uma grande objeção com relação ao uso deste método: ele não obedece a um sistema de camadas, onde a camada mais abaixo não pode fazer chamadas a camadas mais acima, o que o *upcall* viola.

Concorrência e Comunicação entre Processos(IPC)

Podemos definir concorrência como uma condição de disputa (corrida), onde dois ou mais processos querem ter acesso simultâneo à memória compartilhada. Esta região compartilhada é chamada de região crítica. Para solucionar estas regiões, existe quatro condições que devemos obedecer para que a exclusão mútua – impedir que outros processos utilizem variáveis ou arquivos que estejam sendo utilizados por um processos – ocorra:

- Nunca dois processos simultaneamente em uma região (uma *thread* toma conta da região crítica e não sairá enquanto estiver dentro);
- Não se pode considerar velocidade ou número de CPUs (ou seja, os processos que forem mais rápidos não se prevalecem sobre os mais lentos);
- Nenhum processo executando fora de sua região crítica pode bloquear outros processos;
- Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Uma das implementações de exclusão mútua é a espera ociosa, onde, caso um processo X tenta entrar em uma região crítica já ocupada, este espera sua vez sem fazer absolutamente nada de útil. Um modo de implementação deste tipo exclusão mútua é impedir que o processo seja interrompido ao entrar na região crítica, seja por tempo de relógio ou de qualquer outro tipo. No entanto, este modelo é mais

eficiente para operações realizadas dentro do próprio sistema operacional com um único *core* (como a troca de contexto – estudaremos mais tarde). Mas para processos de usuário é bastante perigoso dar-lhe tanto poder, e, com mais de um *core*, outros *cores* não serão afetados pela quebra de interrupção e poderão acessar a região crítica.

Outra solução seria criar uma chave que decide de quem é a vez, e quando um processo sai de sua vez, este indica a chave para alguém. No entanto, este tipo de solução viola a regra de número 3 – Nenhum processo executando fora de sua região crítica pode bloquear outros processos – pois, se a chave for transferida para um processo que não está requisitando a região crítica, este deixará os outros processos em espera (e pode ser eterna).

Uma abordagem muito mais eficiente e utilizada é utilização de um *lock*(*mutex*), onde o processo que entrou na região crítica primeiro indica que esta está trancada e está indicação é uma operação atômica (ou seja, o barramento de acesso para este local de compartilhamento é impedido para outros componentes do sistema). Vejamos o que acontece com o *mutex*:

1. Criar e inicializar a variável *mutex*;
2. Várias *threads* tentam bloquear o *mutex*, mas somente uma consegue;
3. O vencedor faz o que tem de fazer;
4. Após terminar, larga o *mutex* para que outros tentem;
5. Outro processo ou *thread* pega o *mutex* e repete o processo;
6. Ao término de tudo, o *mutex* é destruído.

Apesar desta solução ser bastante eficiente, ela contém falhas, pois, caso um processo de alta prioridade inicie uma espera ocioso, este ficará sempre sendo executado e não permitirá que o de baixa prioridade que se encontra na região crítica a libere. Além disso, como é de se imaginar, como a espero é ociosa, a CPU fica gastando seus esforços com verificar se a trava já foi desbloqueada.

Problema do Produtor-Consumidor

Para resolver os problemas acima foi criado um mecanismo que coloca os processos para o estado de bloqueado / desbloqueado em vez de entrar em espera ociosa. O problema do produtor-consumidor ilustra bem este esquema.

Este é um problema básico de computação, onde existe uma *thread* que produz certo dados e existe outra *thread* que consome estes dados. No entanto, ambas podem ter velocidades diferentes e isso pode levar a dois fatos: ou alguém vai produzir mais do que se pode consumir (havendo perda de dados na memória limitada) e se caso se consuma demais, consumidor fica sem ter o que consumir.

Para a solução desse problema, podemos usar uma solução bem simples, a solução dormir e acordar, onde, quem esteja acelerado demais irá dormir (esperar um pouco) e irá acordar o outro lado quando produzir algo. Lembrando que mesmo assim, temo aqui uma variável crítica que é o contador de itens no *buffer*, logo, é necessário que a adição e a remoção de itens neste *buffer* seja atômica para que não haja problemas no contador.

Semáforo

Semáforo é uma variável que tem como função o controle de acesso a recursos compartilhados, dizendo o valor do semáforo a quantidade de processos ou *threads*

que podem acessar o recurso compartilhado. No caso de exclusão mútua, apenas um processo por vez, logo seu sinal é 1.

Suas principais operações são:

- Inicialização: valor do semáforo definindo a quantidade de processos com acesso ao mesmo tempo;
- Operação *wait*, *down* e *p*: decrementa o valor do semáforo. Em caso de valor zerado, o processo é posto para dormir;
- Operação *signal*, *up* ou *v*: se o semáforo estiver com o valor zero e existir algum processo adormecido, um processo será acordado. Caso contrário, o valor do semáforo é incrementado;

Estas operações de incrementar e decrementar devem ser operações atômicas, ou indivisíveis, ou seja, não podem ser interrompidas no meio (para que as operações terminem com os devidos valores).

Um caso de uso efetivo é semáforos é um *deadlock*, quando um processo pode causar que outro seja interrompido.

Monitor

O monitor é alguém responsável por supervisionar os processos ou *threads*, dando vez para aquele que ele acha que está na vez. O problema do produtor-consumidor é um caso que pode utilizar monitor, onde o produtor e o consumidor não se importam com a implementação da resolução do problema, eles apenas cuidam de produzir ou consumir.

Podemos dizer que o monitor abstrai a função de cada operário com relação impedir que alguém consuma demais ou alguém produza demais. No entanto, vale ressaltar que monitores é um conceito de linguagem de programação e nem todas as linguagens implementam o seu compilador com tal sistema, de modo que torna obsoleto seu uso.

Variáveis Condicionais

Apesar de usarmos um mutex para impedir que outras *threads* entrem em regiões críticas, isso pode não ser suficiente, existem casos que outras condições são necessárias. Este é o caso do produtor-consumidor, posso impedir o produtor de produzir, mas, caso não tenha espaço para gerar novos elementos, ele não pode gerar elementos novos. Este é um mecanismo complementar a *mutex*.

Deadlock

Já ouvimos falar deste falha de concorrência que pode ocorrer em muitos casos. Justamente por isto, iremos estudar ela com maior profundidade. Vamos primeiro entender melhor a interação dos processos.

Níveis de Interação de Processos

- Processos Independentes: Não existe troca de informações;
- Processos Cooperativos (comutativos): Um processo produz informações utilizadas por outros processos;
- Processos Competitivos: Competem pelo uso do mesmo recurso.

Agora vamos imaginar que um certo processo detenha um recurso A e solicite um recurso B, enquanto que outro recurso detenha B e solicite A. Ambos processos ficarão bloqueados enquanto esta situação não for resolvida. Este é um típico cenário de *Deadlock*. Este ocorre quando processos tem acesso exclusivo aos dispositivos e são recursos não preemptíveis (ou seja, o recurso não pode ser retirado do processo sem causar falha).

Condições de Ocorrência de *Deadlocks*

Existem quatro condições para que se ocorra um *Deadlock*, e só ocorrerá caso todos ocorram:

- Condição de exclusão mútua: um recurso está em um dos dois estados, ou associado a um processo ou disponível;
- Condição de Posse e Espera: Processos que retêm recursos em um determinado instante podem requisitar novos recursos;
- Condição de não preempção: Recursos concedidos previamente a um processo não podem ser tomados do mesmo, devem ser liberados por vontade do processo;
- Condição de espera circular: Deve existir uma cadeia circular de dois ou mais processos onde cada membro espera pelo recurso destinado para o membro seguinte da cadeia.

Estratégias para tratar *Deadlocks*

Existem quatro maneiras de se tratar um Deadlock:

- Ignorar completamente o problemas: Este é o que o Windows e o Linux fazem. Caso um Deadlock ocorra, a aplicação fica responsável por trata-lo e não o sistema operacional.
    - Ideal para sistemas onde Deadlock raramente acontecem e o custo de prevenção é muito alto.
- Detecção e recuperação: É feita uma observação da posse e solicitações dos recursos (encontrar um Deadlock dentro de um grafo de posse e solicitações). Para este tipo, há três recuperações possíveis:
    - Recuperação através de preempção: Retirar o recurso de algum outro processo, o que fica a depender da natureza do recurso.
    - Recuperação através de reversão de estado: Verifica o processo periodicamente, usa o estado salvo para, quando o processo entra em Deadlock, o processo é reiniciado para um estado anterior.
    - Recuperação através da eliminação de processos: uma forma grosseira de quebrar um Deadlock, eliminando um dos processos do ciclo, escolhendo aquele que pode ser reiniciado desde o início.

- Alocação segura dos recursos de modo a evitar que *deadlocks* ocorram: Existem algoritmos que fazem está verificação, mas são bastante complicados.
- Prevenção: Temos de fazer que uma das quatro condições necessárias para Deadlock não ocorra de maneira alguma.
    - Atacando condição de exclusão mútua: Podemos evitar alocar recursos quando ele não for absolutamente necessário, diminuindo o número de processos que possam de fator requisitar o recurso. Este método é utilizado na impressora com o esquema de *spool* (fila).
    - Atacando condição de Posse e Espera: Podemos exigir que os processos requisitem todos os recursos que iram precisar antes de iniciar (jamais espera pelo que precisa). No entanto, isto é meio que difícil de implementar pois alguns processos podem não saber quais e quantos recursos irá precisar no início de execução e também retém recursos que outros processos poderiam estar usando. Uma variação é o processo abdicar de todos os seus recursos atuais para conseguir todas aqueles que são necessárias imediatamente.
    - Atacando a Não preempção: Retomar os recursos alocados ou criar uma virtualização do recursos em disco (mas nem todos os recursos podem ser virtualizados, como o S.O.);
    - Atacando a Espera Circular: Ordenar numericamente os recursos de modo que nenhum processo pode tomar algum recurso de número menor do que ele já contêm.

Bloquei de Duas-Fases

É um bloqueio de recursos que acontece em duas etapas distintas, uma com alocação de recursos e outra com atualização e liberação de recursos.

Fase Um (*Growing Phase*): Processo tentar bloquear todos os recursos de que precisa, um de cada vez. Caso o recurso que se deseja bloquear, já estiver bloqueado, este reinicia a sua fase de crescimento, liberando todos os recursos adquiridos.

- Fase Dois (*Shrinking Phase*): Dado que a fase um conseguiu bloquear todos os recursos necessários, há a execução do que se deseja fazer com a respectiva liberação dos recursos.

Existe um tipo de Transação Especial que são consideradas por *Strict two-phase lock*. Neste caso, é necessário ter todos os recursos necessários para fazer fase dois e apenas ao final da fase dois ele libera todos os recursos de uma vez só. Caso contrário, o processo é abortado para não prevenir erros.

Para controle de Concorrência no modelo de duas-fases, o sistema operacional tem duas implementações:

- Usar *locking* em ordenação *timestamps*: o sistema operacional detecta conflitos entre processos a cada acesso a recursos, onde, se haver conflito, a transação é interrompida imediatamente. É um visão otimista.

- Abordagem Otimista: o sistema operacional permite que o processo prossiga até o término para checar se foi bem sucedida (sem conflito). Em caso de conflito, transação é refeita.

Escalonamento

Já falamos bastante de escalonamento, o mecanismo que decide quem irá ser executado naquele instante. No entanto, foi algo bem superficial. Iremos agora aperfeiçoar nosso conhecimento sobre escalonamento e seus mecanismo.

Para iniciar, vamos apenas dizer que há quatro tipos de sistemas operacionais: Monotarefa, multitarefa, monousuário, multiusuário. Apesar do tipo, em todos eles tem de pensar em como evitar quem um processo monopolize o sistema: já sabemos da solução: sistema de compartilhamento de tempo, ótimo para mecanismos com I/O-bound. Lembrando que um processos pode ser terminado por três razões: término do processo em si, comando para dispositivo de entrada e saída ou seu tempo acabou.

Multiprocessamento

Temos a noção de multiprocessamento agora, mas o que acontece na máquina? Há uma troca de processos, que consiste em trocar o valor dos registradores de contexto da CPU. Ou seja, um novo processo é escalonado, então um novo contexto é CPU é iniciado. Mas o que é necessário para haver multiprocessamento? Tem de haver suporte de duas entidades:

- Suporte de Hardware:
  - Noção de temporizador;
  - Habilidade de interrupção: Sem esta habilidade, a CPU teria de ficar esperando a execução do comando de entrada/saída. Com ele, este pode realizar outras tarefas enquanto não for interrompido pelo dispositivo;
  - Proteção de memória contra outros processos;
- Suporte de S.O.:
  - Habilidade de escalonamento (fundamental);
  - Alocação de memória;
  - Gerenciamento de periféricos (dispositivos do sistema);

Processo de Interrupção

Por interrupção ser uma arma muito útil no favorecimento de multiprocessamento, temos de entender como ele ocorre na máquina. Ele ocorre em duas fases, primeiro no hardware e, posteriormente, no software.

1. Hardware:
   a. Dispositivo pede interrupção;
   b. Processador salva no registrador PSW o status do programa/processo a ser interrompido;
   c. Processador carrega novo valor de PC baseado na interrupção;
2. Software:
   a. Salva resto da informação do contexto do processo;

b. Processa rotina de Interrupção;
c. Restaura informação do estado do processo em execução antes da interrupção;
d. Restaura o registador PSW e PC do processo em execução antes da interrupção;

Filas de Escalonamento

Existem três filas de escalonamento, a depender do que vai acontecer:

- Fila de alto-termo: Decide quantos programas serão admitidos no sistema, alocando memória e criando um processo para ele.
- Fila de curto-termo: Decide qual processo deve ser executado. Famosa fila de escalonamento.
- Fila de Entrada/Saída: Decide qual processo (com E/S) pendente deve ser tradado pelo dispositivo de E/S.

Algoritmos de Escalonamento

Existem três tipos de categorias de Escalonamento:

- Em lote (batch): Processos que dependem muito mais da CPU (*CPU-bound*) e informam um processo como um todo, dizendo quando e quanto tempo deverá ser executado. Existem dois tipos de processos em lote, os com interrupção e os sem.
- Interativo: Processos que dependem muito mais de dispositivos de entrada e saída (*I/O-bound*) e utilizam muito mais o usuário como seu fornecedor.
- Tempo real: Processos de tempo real, ou seja, tempo é uma variável de corretude, diferentemente dos outros dois, que tempo é uma variável de eficiência. As ações devem ser realizada no tempo em que devem ser realizadas, não importando se vai demorar ou não, mas sim, se obedeceu o tempo estimado para ele.

Critérios dos Algoritmos De Escalonamento

Bem, não adiante criar qualquer algoritmo e achar que ele fará o trabalho. É necessário que ele obedece a alguns critérios, que também dependem de sua categoria. Vejamos os critérios gerais para todo e qualquer sistema:

- Justiça: dar a cada processo uma porção justa da CPU;
- Aplicação da política: verificar se a política de escalonamento está sendo obedecida;
- Equilíbrio: Manter ocupadas todas as partes do sistema;

Vejamos agora os critérios de cada categoria:

- Sistema em batch (lote)
  - Vazão (throughout): maximizar o número de Jobs (tarefas) por hora;
  - Tempo de Retorno: minimizar o tempo dentre a submissão e o término;
  - Utilização de CPU: manter a CPU ocupada o tempo todo;
- Sistemas interativos:
  - Tempo de resposta: responder rapidamente as requisições;

    - Proporcionalidade: satisfazer as expectativas dos usuários;
- Sistemas de tempo real:
    - Cumprimento dos prazos: evitar a perda de dados;
    - Previsibilidade: evitar a degradação da qualidade em sistemas multimídia;

Escalonamento Preemptivos

Existem dois tipos de mecanismo de escalonamento, um deles é o escalonamento preemptivo (o outro seria o não-preemptivo), que permite a suspensão temporária de um processo após seu *quantum* ou *time-slice* (tempo de duração de execução) tenha terminado.

No entanto, este sistema apresenta uma imensa dificuldade, a Troca de Contexto: Mudar de um processo para outro requer um certo tempo para a administração salvar e carregar registradores e mapas de memória, atualizar tabelas e listas de SO, entre outros fatores.

Para isso, teríamos uma solução que é aumentar o *quantum* de processo para que o tempo de troca de contexto seja desprezível se comparado ao todo. Mas isto nos levaria a outro problema, o aumento do *quantum* causaria o aumento de espera de outros processos, o que pode ser danoso.

Ou seja, enquanto que um *quantum* muito pequeno causa um excesso de *overhead* e de trocas de contexto, mas um *quantum* muito grande pode causar uma impressão de lentidão para os outros processo. Temos de escolher um *quantum* que otimize estes dois lados ou que satisfaça a nossa necessidade.

Algoritmos de Escalonamento

Vamos agora estudar alguns exemplos de algoritmos de escalonamento, onde são utilizados, suas vantagens e desvantagens.

- Earliest Deadline First (EDF): Algoritmo de escalonamento utilizado por sistemas de tempo real. Algumas características são:
    - Preemptivo;
    - Considera o momento em que a resposta deve ser entregue;
    - Processo se torna mais prioritário quanto mais próximo do deadline (prioridade dinâmica);
    - Algoritmo Complexo.
- First-In First-Out (FIFO): Algoritmo de escalonamento utilizado por sistemas em lote, utilizando listas de processos sem prioridade, com simplicidade e justiça. Considera para sistemas não-preemptivo.
- Shortest Job First: Também é um algoritmo de escalonamento para sistemas em lote não-preemptivo visando colocar em execução o job (trabalho) que tiver menor tempo de execução. Apesar de proporcionar o melhor tempo de retorno, é um algoritmo não-justo pois pode causar estagnação (*starvation*) – enquanto processos menores que ele estiver aparecendo, ele não será executado. Apesar disto, pode ser resolvido com prioridade dinâmica.
- Shortest Remaining Job Next: Também para sistemas em lote, exceto que é utilizado em preemptivo, dando prioridade para aqueles que tem menor tempo sobrando para término.

- Round-robin: Utilizado para sistemas interativos, possui uma lista de processos executáveis onde, cada vez que o *quantum* de um processo acaba, este vai para o final da fila e cede para o próximo da fila.
- Prioridade: Também para sistemas interativos, possui classes de prioridade, onde os de prioridade mais alta são executados primeiro para, posteriormente, os de prioridade mais baixa serem executadas. Não-justo pois, caso as filas acima do processo não esvaziarem, este jamais será executado.
- Multiple feedback Queue (Híbrido): O escalonamento tem dois níveis, um para os processos em lote –utilizando partições de lote-, outro para processos interativos. Processos interativos são ativados assim que requisitados enquanto que processos em lote ficam em espera pela liberação da partição do lote. O MFQ utiliza abordagem de prioridades dinâmicas com adaptação baseada no comportamento de cada processo. Algumas regras são:
    - Novos processos entram na primeira fila de prioridade;
    - Quando o *quantum* de um processo acaba, desce a prioridade;
    - Se requisitar E/S, sobe prioridade (pois E/S-bound são prioritários).
- Escalonamento garantido: É um modo de dar ao processo ou aos usuário uma fatia de tempo em CPU relativa ao número de processos /usuário ativos naquele momento. Ou seja, é uma fração do todo. Em casos em que algum elemento utiliza mais CPU do que deveria (ou menos CPU), este processo é então rebaixado de categoria (ou aumentado). É de difícil implementação.
- Escalonamento por loteria: é dato um "bilhete" da cada processos e, de modo aleatória, o S.O. seleciona um dos bilhetes para ser executado. Dado mais bilhetes a processos que são mais importantes. E de modo análogo, quanto mais bilhetes um processo tiver, mais tempo de CPU ele ganha.
- Escalonamento por fração justa: A cada usuário é dado uma fração justa de tempo de CPU que independe da quantidade de processos que este conter.

Escalonamento de Threads

Threads, como já sabemos, são processos mais curtos, que possibilitam multiprogramação. Existem duas formas de escaloná-lo, a depender da implementação:

- Threads de Usuário: O escalonador do SO não reconhece threads e por isso se limita a escalonar o processo, enquanto que o processo que decide o escalonamento de suas threads.
- Threads de Núcleo: O escalonador do SO reconhece threads e, por isso, escalona threads, em vez de escalonar entre processos.

Gerenciamento de Memória

Em um mundo ideal, o que todo programador deseja é ter acesso a uma memória grande, rápida e não-volátil. Sabemos que isso não é verdade e que existe uma hierarquia de memória que define bem essas características:

- Pequena quantidade de memória, rápida e de alto custo - cache;
- Quantidade considerável de memória principal de velocidade média e custo médio;
- Gigabytes de armazenamento em disco de velocidade e custo baixos.

Podemos então definir que a troca de acesso a essas memórias e a disposição dos dados nelas é que define um bom gerenciamento de memória, de modo a aumentar a velocidade do sistema como um todo.

Recolocação e Proteção

Não se sabe como certeza onde o programa será carregado na memória, ou seja, localizações de endereço de variáveis e de código de rotinas não podem ser absolutos. Para recolocação e proteção dos dados na memória, é usado a ideia de base e limite, onde a base indica onde o programa começa e o limite é o seu tamanho de ocupação na memória. Fazemos isso para dar a cada processo um espaço de endereçamento separado (protegido) – partição. Podemos então definir como espaço de endereçamento um conjunto de endereços que o programa tem permissão para utilizar.

Utilizando o ideia de registrador-base e registrador limite, o software é capaz de traduzir o endereçamento do programa para um endereçamento físico, garantido que estão acessará o local de memória que fora destinado a ele.

A maior desvantagem deste modelo é a necessidade de realizar somas e comparações toda vez que for referenciar a memória, o que deixa o processamento de memória mais lento.

Existem outros modelos mais complexos e rápidos, mas nos limitaremos a este modelo de proteção e realocação.

Troca de Processos e Mapeamento

Já sabemos que processos entram e saem da memória a medida que o tempo passa, ou seja, é necessário fazer um gerenciamento da troca de processos na memória, e com isso uma alteração na alocação da memória. Existe dois métodos para a troca de processos: *swapping*, que consiste basicamente em levar todo o processo para a memória, executá-lo durante um tempo e posteriormente retorná-lo ao disco, sendo salvo o seu contexto em cada troca. Também há o de memória virtual, que veremos em breve, mas é executar o programa com ele parcialmente na memória e parcialmente em disco.

Para que um processo seja armazenado na memória, é relevante levarmos em conta o fato de que muitos programas são expansíveis. Deste forma, é importante garantirmos um espaço "extra" para que seja permitido o crescimento. Com isso em mente, podemos dizer que um processo é alocado com três diferentes partes, um para o armazenamento do programa, um para o armazenamento dos dados e outro espaço para o armazenamento da pilha do programa e entre esses dois últimos teremos ao espaço extra, que permite que os dados e a pilha do programa cresça. Em

caso do tamanho passar dos limites, este é então realocado para outro espaço maior disponível.

No entanto, na troca de processos, pode acontecer que acha "buracos" na memória que não tão sendo preenchidos porque nenhum processo, até agora, coube naquele espaço. É necessário fazer um mapeamento da memória para saber os locais livres e colocar os novos processos nestes espaços. Existem dois modos de fazer tal mapeamento:

- Mapa de Bits: A memória é separada em segmentos e, se um dado segmento está ocupado, o mapa de bits coloca um no seu mapeamento para identificar que está ocupado. Deste modo, os espaço vazios são identificados com zero, e uma sequência de zeros define o tamanho do espaço vazio. Deste modo, sua maior desvantagem é encontrar uma sequência de zero, pois é necessário percorrer o mapa para encontrá-la, sendo um processo bastante lento.
- Lista encadeada: Existe uma lista encadeada, onde cada nó tem um indicador de ocupação por programa ou espaço livre, um registrador-base para dizer onde começa e um registrador-limite para dizer o tamanho do segmento. Deste modo podemos manter uma atualização rápida e simples da lista (principalmente se esta for ordenada)

Com isso, podemos realizar fragmentações (estudaremos posteriormente), para que os buracos sejam evitados e garanta mais espaço livre para os programas.

Algoritmos para alocação de memória

É importa que o algoritmo escolhido para o sistema operacional alocar o programa na memória seja um que convenha com o desempenho da máquina. Vamos estudar três deles:

- *First Fit*: Com a ideia de encontrar o espaço de memória o mais rápido possível, o *First Fit* busca o primeiro espaço de memória livre que encontrar e então coloca o processo neste segmente. Caso o processo seja menor que o segmento livre, o restante é segmentado e alocado novamente como segmente livre de tamanho menor.
- *Next Fit*: Apesar de seguir a mesma ideia do *First Fit*, o *Next Fit* tem o diferencial de continuar a busca do último local de memória que parou da última vez que alocou um processo.
- *Best Fit*: Ao contrário do *First Fit* que aloca o processo no primeiro lugar vago, o *Best Fit* procure em toda a lista o menor lugar disponível para aquele processo, de modo a minimizar a quantidade de sobras. Pelo fato de ter de procurar por toda a lista em cada alocação, o *Best Fit* é um algoritmo muito mais lento.
- *Worst Fit*: Segue o mesmo princípio que o *Best Fit*, exceto que procura o espaço de maior quantidade disponível.

Todos os algoritmos poderiam ter melhor desempenho caso mantivéssemos lista distintas para segmentos ocupados e livres. No entanto, o ganho no desempenho do algoritmo, gera uma perda na simplicidade e aumento na redução da liberação de memória, pois, ao ser liberara, este deveria voltar a lista de segmentos livre e concatenar-se com as outras, de modo a gerar um segmento livre maior. Caso isto não aconteça, o que teremos é uma rápida fragmentação da memória.

Multiprogramação com Partições Fixas

Já sabemos que partição (espaço de endereçamento de memória) é um local da memória destinado a um programa. No entanto, para oferecer multiprogramação, podemos realizar dois tipos de entrada de processos nas partições:

- Filas de entradas separadas para cada partição: Existem várias filas e cada uma possui os processos que cabem naquela partição e que se limitam pelo seu tamanho máximo. Deste modo, evitamos alocar processos em partições de grande capacidade e subutilizar o espaço de memória
- Fila única de entrada: Os processos vão entrando nas partições a medida que chegam. Apesar de veloz, acaba por subutilizar a memória pois podemos acabar ocupado um partição da memória com um programa muito pequeno.

Memória Virtual e Paginação

Quando um programa acaba por não caber na memória, este deve ser seu espaço de endereçamento quebrado em páginas para então poder executar na memória. O ideia é de que nem todas as páginas precisam estar na memória simultaneamente para que o processo ou programa rode. Deste modo, apenas as páginas em uso serão carregadas.

Em um mundo ideia de código sequencial sem pulos, as páginas são colocadas na memória a medida que o código avança. No entanto, no mundo real, temos que carregar na memória o número máximo de páginas que conseguimos, e, caso uma página chame uma página não carregada, o S.O. deve escolher a melhor página que deve ser retirada para dar local a página chamada.

Já memória virtual é um espaço em disco que simula a memória principal, ou seja, na verdade, o que acontece com um programa muito grande é que é feito uma cópia do programa para a memória virtual, neste local é quebrado em página e então as páginas requisitas são carregas na memória.

Você deve estar se perguntando então por que fazer isso se a memória secundária é lenta para acesso? Bastante simples a resposta: como o disco é divido em seções e trilhas, as seções mais ao centro são lidas mais rápidas e é neste local que é colocado tanto o sistema operacional quanto a memória virtual (visto que não cabem na memória principal).

Para que a ocorra o acesso as páginas não carregadas na memória, a CPU fornece à Unidade de Gerenciamento de Memória (*MMU* - unidade de hardware) um endereço de memória, verifica se é um endereço real ou virtual. Em caso de real, acessa a memória diretamente, em caso de virtual, acessa o disco para carregar a página no espaço ocupado pelo processo. Vale salientar que a MMU é um elemento de hardware, por este motivo ela é efetivamente mais rápida e seus cálculos são feitos rapidamente para que ninguém perceba.

A MMU possui uma tabela de páginas que indica onde a página está (se na memória ou se em disco), é fornecido a qual página devemos olhar, a página nos fornece em que local de disco iremos procurar, e então a página pode ser carrega na memória, caso esta não esteja na memória real.

Uma entrada típica de uma tabela de páginas consiste de 5 campos:

- Número da moldura de página: é o espaço em memória no qual a página se encontra; consiste de um range de bytes e um valor deste range é onde a página se encontra;
- Presente/ausente: bit que indica presença na memória real;
- Modificada: bit que indica se página foi modificada recentemente;
- Referenciada: bit que indica se página foi lida recentemente;
- Cache desabilitado;
- Proteção.

Acelerando a Paginação

Como o acesso a memória em disco é lenta, é necessário fazer o mapeamento de endereço virtual para endereço físico rapidamente. Se o espaço de endereçamento virtual for grande, a tabela de páginas será grande e varrê-la seria bastante enfadonha para o computador, deixando o procedimento lento.

Para solucionar este problema, foi criado um memória associativa ou TLD (*Translation lookaside Buffers*), onde ficam registrados os endereços referenciados mais recentemente, podemos considerar que é um cache para tabela de páginas. Apesar de acelerar o procedimento, é em troca de uma ocupação de espaço na memória.

Outra solução proposta é o esquema de Tabelas de Páginas de Multi-níveis, ou seja, uma tabela referencia outras tabelas de modo que só temos que percorrer as pequenas tabelas e que levam a outras tabelas, em vez de uma grande e única tabela.

Substituição de Páginas

A situação para que haja necessidade de substituir páginas é a seguinte: Uma dada página é chamada, mas ela não está na memória real (*page-fault*) e, como todos os espaços de páginas estão ocupados, é necessário retirar alguma páginas para colocar a chamada. Mas a pergunta é, qual página deve ser retirada? Esta é a grande pergunta da substituição de páginas, não podemos remover qualquer uma, pois isso poderia levar a uma excesso de substituição, o que leva a uma sobrecarga do sistema.

Vale salientar que, se uma dada página foi modificada e ela será apagada da memória principal, ela deverá ser salva na memória virtual para não perder os dados modificados. Além disso, um ponto extra é que nunca é aconselhado remover uma página que está sendo muito utilizada, pois, provavelmente, você irá utilizá-la posteriormente.

Os algoritmos que estudaremos agora será baseados no algoritmo ótimo, uma algoritmo ideal que procura substituir o mais tarde possível uma página, sua ideia é retirar uma página sabendo que ele só será utilizada o mais tarde. Já se vê que é impraticável pois, se seres humanos não preveem o futuro, máquinas muito menos.

- FIFO (primeira a entrar, primeira a sair):Mantém uma lista encadeada de todas as páginas, quanto mais próximo da cabeça da lista, mais antiga ela é. Páginas são removidas da cabeça e novas páginas adicionadas ao final da lista. Sua maior falha é que pode ser que a página mais antiga seja usada com muita frequência, mas ainda assim será removida. Mas é bastante simples de implementar no mais puro modo.

- Segunda Chance (SC): Muito semelhante a FIFO, no entanto, se uma dada página foi referenciada e estava prestes a ser removido, esta ganha uma segunda chance e vai para o final da fila, reiniciando o valor do bit R para 0. Apesar de solucionar o problema da FIFO, acaba por ter um processamento maior, porque tem de percorrer toda a lista tendo de procurar uma página ainda não referenciada.
- Não usada recentemente (*NUR,NRU*): Páginas são colocadas em quatro classes (não referenciado e não modificada, não referenciada e modificada, referenciada e não modificada e referenciada e modificada). Tiramos sempre da classe que está mais acima, pois a probabilidade de utilização desta página é menor. Se contiver várias páginas dentro de uma mesma classe, é escolhido uma página aleatoriamente. É um ótimo algoritmo porque é fácil de entender e simples de implementar e próximo do ótimo (adequado).
- Menos recentemente usada (*MRU,LRU*): Retira a página que há mais tempo não foi usada. Contém uma lista encadeada que é atualizada dinamicamente, onde a página mais recente fica na cabeça da lista e a menos, ao final. A lista é atualizada a cada referência da memória. Outra alternativa é ter um contador na tabela de páginas e escolhe página com contador de menor valor, que é zerado periodicamente para evitar que páginas com contadores altos e não mais utilizados fiquem na memória. Apesar de bastante próximo do ótimo, é um algoritmo custoso em termos de processamento (necessita atualização constante) e em termos de hardware, pois é necessário uma infraestrutura que suporte.
- Conjunto de Trabalho (*WS*): Define-se as páginas que estão sendo utilizadas e não devem ser removidas. Quando há uma necessidade de substituição de página, a tabela de página é varrida, verificando o bit de referenciado e também o tempo em que a página foi acessada pela última vez. É calculado uma idade para a página baseada no tempo virtual atual menos o instante em que a página foi última vez referenciada. Se a página não foi referenciada e sua idade for maior do que uma dada constante, está é adequada para sair. Caso contrário, se foi referenciada, seu instante atual fica o tempo virtual atual, e caso não tenha sido referenciada e idade for menor ou igual à dada constante, mantém o menor tempo. Sua maior desvantagem é ficar dando varreduras na tabela inteira para atualizar todos os dados, podendo ser demorado.
- Relógio (*Clock*): As páginas são colocadas em um círculo (como um relógio) e há um ponteiro para a página mais antiga do círculo. Ocorrendo falta de página, se a página onde o ponteiro estiver não tiver sido referenciada, ela é retirada e o ponteiro avança. Caso contrário, seu bit é colocado para zero e o ponteiro continua a procura por um bit 0. Isto permite que não haja inserção de páginas constantemente ao final da lista encadeada.
- Conjunto de Trabalho Clock (*WSClock*): Como o algoritmo de Conjunto de Trabalho tem de varrer a tabela inteira para atualizar seus valores, criou-se este algoritmo para resolver tal problema. Ele utiliza ambos os algoritmos de Conjunto de Trabalho e de Clock, onde o critério de retirada agora é a idade e referencia como no Conjunto de Trabalho e, para não

precisar percorrer todas as páginas, ela começa a analisar até encontrar a página a ser substituída e então o ponteiro avança.

Alocação local versus alocação global

De vez em quando, ao realocar a página, temos de saber se vamos substituir uma moldura que está entre as molduras daquele processo que teve falta de página ou se vamos retirar qualquer moldura que está na memória. O primeiro tipo de substituição se chama Alocação local, onde é definido uma número máximo de moldura por processo e, caso o processo tenha falta de página, ele deve substituir um de suas páginas. Já o segundo é a Alocação global, onde o processe pode substituir qualquer moldura de página.

Podemos dizer que a Alocação global é mais eficiente que a local, pois temos comprovado que o conjunto de trabalho de um processo pode variar aleatoriamente e, caso ele cresça e não haja espaço, haveria uma falta de página constante e, com isso, retardo no processamento.

Tamanho de página

O tamanho da página que será levada a memória é um fator importante na paginação, visto que temos vantagens e desvantagens em casos de páginas de tamanho pequenos ou grandes. Vamos analisa-los:

- Páginas de tamanhos pequenos: Existem dois argumentos a favor das páginas de tamanhos pequenos.
    - Minimização da fragmentação interna: Toda vez que se gera páginas para um processo, é normal de que ele não ocupe um número inteiro de páginas e, em média, é esperado que sobre metade da última página. Deste modo, com n segmentos de código na memória, é esperado n*p/2 espaço de página desperdiçado em fragmentação interna. Logo, quanto menor o espaço da página, melhor.
    - Otimização da memória: quanto menor a página, é mais provável que um processo ou programa ocupe a unidade de página. Isto é bem visto por programas pequenos. Se tiver uma página de 32KB e um programa de 4KB, teremos sempre na memória um programa de 4KB ocupando um espaço de 32 na memória, desperdiçado.
- Página de tamanhos grandes: o fator mais importante para a defesa de páginas grande é a alocação de memória para a tabela de páginas. Como, quanto mais páginas tivermos na memória, maior é a quantidade de entradas na tabela de páginas e, com isso, maior é a tabela de páginas que deve ser armazenada. Além disso, temos também a lentidão de levar páginas do disco para a memória, que normalmente é feita de uma página por vez, então, quanto mais páginas, maior a lentidão. Um número grande de páginas então diminui essa lentidão e também a quantidade de memória alocada para a tabela de páginas.

Claro que temos um número ótimo de tamanho de páginas, que é dado por uma fórmula, calculando sua derivada. Mas não entraremos em detalhes, apenas podemos dizer que, atualmente, temos um tamanho de página de 4KB ou 8KB, e o crescimento aumenta (lentamente), com o aumento do tamanho da memória.

Segmentação

Como já sabemos, quando um processo é colocado na memória ele inclui várias tabelas - a de chamada, a de árvore sintática, a tabela de constantes, o próprio código-fonte e a tabela de símbolos. Se dermos um espaço de endereçamento único para o processo como um todo, pode ser que uma dessas tabelas cresça demais e ocupe o espaço máximo que recebeu, impedida de expandir pois outros programas ou outras tabelas do mesmo programa estão ocupando o espaço acima.

Para permitir então que as tabelas dos processos possam ser registradas na memória de modo a garantir a expansão e independência de espaço, foi criada a ideia de segmento, um micro espaço de endereçamento dado a cada tabela do programa ou processo de modo a garantir sua expansão dentro do seu espaço de endereçamento. E, para garantir que a tabela não ultrapasse seu tamanho máximo de segmento, são dados segmentos com tamanhos grandes o suficiente para aquele propósito. Para então acessar um endereço em uma memória segmentada é dado um par que indica o número do segmento e um endereço nesse segmento.

Além de simplificar o tratamento a estrutura de dados que estão sempre crescendo, a segmentação também trás uma independência das rotinas do programa. Em uma memória unidimensional, quando uma rotina é alterada, todas as outras terão de ser reorganizadas no espaço de endereçamento, pois todas compartilham o mesmo. Já na segmentação, como cada rotina possui seu espaço de endereçamento, quando uma rotina é altera, unicamente seu espaço de endereçamento será alterado (se necessário), não influenciado as outras, o que diminui o tempo de processamento.

Concomitantemente, a segmentação oferece suporte ao compartilhamento de dados e rotinas. Visto que cada uma tem seu próprio espaço de endereçamento, o segmento é compartilhado por cada processo que o utiliza, de modo que não é necessário fazer uma cópia daquele espaço de endereçamento para o processo.

Mas, assim como a paginação, a segmentação também oferece fragmentação, a chamada fragmentação externa. Quando processo maiores liberam a memória e processos menores então, acaba por gerar um uma lacuna dentro da memória, deixando a mesma dividida em regiões, algumas ocupadas e outras não. Este fato pode ser corrigido com o uso de compactação.

Estudo de Caso: MULTICS

Para utilizar a vantagem dos dois mecanismos estudados ao longo desse capítulo – segmentação e paginação – foi desenvolvido o MULTICS, um mecanismo que oferecia segmentos paginados, ou seja, cada segmento é tratado como memória virtual, o que fazia com que nem todo o segmento estivesse na memória quando executado (apenas a parte utilizada), o que garantia efetivo uso de espaço da memória, além de facilitar a programação, gerar modularidade, proteção e compartilhamento simples com o uso de segmentos.

O MULTICS utiliza várias tabelas, uma delas é a de descrito de segmento, que armazena os segmentos atuais. Esse segmento aponta para outras tabelas que indica as páginas daquele segmento. Deste modo, para se acessar uma palavra em uma página de um certo segmento, é necessário fornecer o número do segmento a ser procurado, o número da página no segmento e o deslocamento dentro desse página. Com isso, o endereço MULTICS consiste de duas partes: o número do segmento como a primeira parte e o endereço dentro do segmento – que consiste do número de

página e o deslocamento dentro da página. Com isto podemos acessar o endereço de memória real.

Sistemas de Arquivos e Entrada e Saída

Descendo um nível na hierarquia de memória dos computadores, iremos agora entender como funcionam os sistemas de arquivos do computador de modo a possibilitar o armazenamento de uma grande quantidade de informação de modo persistente (sobreviver ao término do processo) e possível de compartilhamento, onde processos concorrentes podem ter acesso a essa informação.

Concomitantemente, iremos estudar a obtém de dados e sua exibição, ou seja, a entrada e saída de dados através dos periféricos.

Introdução a Arquivos

Podemos começar dizendo que arquivos é uma implementação que possibilita a estruturação e organização das informações "jogadas" no disco. Ou seja, o sistema de arquivos é basicamente uma abstração do sistema operacional de acesso aos dados no disco. No entanto, temos uma característica peculiar neste sistema é que ele deve ser uniforma, melhor explicando, este deve ser independente do dispositivo de armazenamento (independe do hardware).

Uma melhor explicação de Arquivo pode ser dito com um conjunto de registros definidos pelo sistema de arquivos, podendo ser armazenado em diferentes dispositivos físicos, constituído de informações logicamente relacionadas, podendo representar programas ou dados.

Além disso, a representação de informação nos computadores é um conjunto de números, o qual, no Windows e no Linux, são chamados de Streams de Bytes. Não por menos, para identificar um arquivo de outro, é necessário dar-lhes nomes, contendo ou não uma extensão (tipo) para aquele arquivo.

Organização de Arquivos

Podemos dizer que a Organização de Arquivos é o modo como os dados estão armazenados internamente. Para isso, se utiliza uma forma simples de organização dos arquivos: uma sequência de bytes não-estruturada de bytes, onde não existe nenhuma estrutura lógica para os dados, ficando a aplicação responsável por defini-la. Como grande vantagem tem-se a flexibilidade para a criação de estrutura de dados. No entanto, existem três tipos de organização implementadas: sequencial, relativa e indexida.

Além disso, existem dois tipos de arquivos, os executáveis e os de repositório, cada um contendo seu próprio formato, diferenciando os executáveis dos de repositório por um "número mágico", todo arquivo executável possui em seu cabeçalho o número mágico.

Métodos de Acesso

Não basta apenas armazenarmos os arquivos, se não pudermos acessá-los. Logo, foram criados métodos de acesso a esses arquivos, a depender de como o arquivo está organizado. Existem três métodos de acesso a arquivos:

- Acesso sequencial: utilizados para arquivos armazenados em fitas magnéticas, limitando o acesso à leitura na ordem em que foram escritas, gravando dados apenas no final da fita. Podemos combinar o acesso sequencial com o direto de forma a acessar diretamente um arquivo e depois percorrê-lo em forma sequencial.
- Acesso direto: permite a leitura e gravação de um registro diretamente em sua posição, utilizando o número de registro, não existindo restrição à ordem em que os registros são lidos e gravados – contrário do sequencial. No entanto, isto só é possível em registro de tamanhos fixos.
- Acesso indexado ou por chave: O arquivo contém uma área de índice onde existem ponteiros para os diversos registros e a partir desta informação realiza-se um acesso direto. Bastante parecido com um hashing.

Operações de Entrada e Saída

Assim como qualquer outra abstração dos sistemas operacionais, o sistema de arquivos oferece chamadas ao sistema que permite às aplicações realizarem operações de entrada e saída, como tradução de nomes em endereços, leitura e gravação de dados e criação e eliminação de arquivos. As chamadas aos sistemas do sistema de arquivos têm a função de oferecer uma interface simples e uniforme entre a aplicação e os diversos dispositivos de armazenamento.

Além disso, todos os arquivos gerados são gerados com atributos, mas que variam de acordo com o sistema operacional. Os mais comuns são o tamanho, criados, proteção, data de criação e modificação, entre outros. O usuário também pode modificar alguns atributos, outros ficam pelo próprio sistema operacional de alterá-los ou não.

Diretórios

Pode-se dizer que diretório é uma maneira mais efetiva de organizar os arquivos dentro de um disco. É basicamente uma estrutura de dados que contém entradas associadas aos arquivos onde estão as informações como localização física, nome, organização e outros atributos. Quando um arquivo é aberto, o S.O. procura a sua entrada na estrutura de diretórios, armazenando as informações do arquivo numa tabela mantida na memória principal, que mantém salva todos os arquivos abertos para aumentar o desempenho das operações em arquivos. Vamos agora analisar alguns tipos de implementação de diretórios de arquivos:

- Nível único: é a implementação mais simples, onde existe apenas um único diretório contendo todos os arquivos do disco. Podemos dizer que é uma implementação bastante limitada já que não permite que usuários criem arquivos com o mesmo nome (já que estariam no mesmo diretório).
- UFD (*User File Directory*): para cada usuário existe um diretório particular e assim cada usuário para tem seus próprios arquivos com seus nomes. No entanto, deve haver um diretório mestre *(MFD*) que é indexado pelo nome de usuário e cada entrada aponta para o diretório pessoa. Podemos dizer que é basicamente uma árvore de caminhos único, com a raiz sendo

a MFD, os ramos a UFD e as folhas, os arquivos, sendo necessário especificar seu diretório para acessar um arquivo.
- Estrutura em Árvore (*Tree Structured Directory*): Adotado pela maioria dos sistemas operacionais e é logicamente melhor organizado. Possibilita a criação de quantos diretório desejar-se, permitindo que diretórios tenham outros diretórios ou arquivos. Como a UFD, cada arquivo possui um caminho único (*path*).

Alocação de espaço em disco

A criação de arquivos exige que o S.O. tenham noção dos blocos ou áreas livres em disco. Normalmente este controle é feito de uma tabela chamada de mapa de bits (já estudada), onde cada entrada da tabela é associada a um bloco e representado por um bit que verifica se o bloco está ocupado ou livre. No entanto, apesar de ideal, é uma estrutura que gera um gasto excessivo de memória, já que para cada bloco existe uma entrada na tabela.

Com isso, existe uma outra forma mais econômica que é realizar o controle por meio da ligação encadeada de todos os blocos livres e cada bloco deve possuir uma área reservada para armazenamento do endereço próximo (uma lista encadeada). Com isso, a partir do primeiro bloco, podemos ter acesso sequencial aos demais de forma encadeada. No entanto, sua maior falha é que a busca de espaço é feita de forma sequencial.

Outro solução mais complexa é a que considera que blocos contíguos são alocados ou liberados de simultaneamente, o que deixa possível manter uma tabela com o endereço do primeiro bloco de cada segmento e o número de blocos livres contíguos que se seguem. A sua vantagem é que o acesso é bastante simples tanto para a forma sequencial tanto para a direta e com agilidade. Já sua maior desvantagem é que para alocar um novo arquivo de tamanho n necessita que haja uma cadeia com n blocos dispostos sequencialmente no disco, além do problema da extensão, pois um arquivo pode se estender e a alocação em blocos contíguos não aceita bem este fato.

Além disso, na hora de alocar um novo arquivo em alocação contígua, é necessário que o S.O. escolha o melhor local para inserir o novo arquivo no disco. Existem três estratégias para isto:

- *First-Fit*: Basicamente o que o nome diz, o primeiro espaço livre encontrado que cabe o novo arquivo será o destinado para o arquivo. A busca, no caso, é sequencial, sendo interrompida ao se encontrar o espaço em aberto. Maior vantagem: mais rápido.
- *Best-Fit*: Seleciona o menor segmento livre disponível que cabe o arquivo. A busca é feita em toda a lista, para que se encontra o melhor, a não ser que a lista esteja ordenada por tamanho (e o tamanho seja sempre mantido ordenado). Maior vantagem: menos desperdício.
- *Worst-Fit*: o maior segmento é alocado, se fazendo, novamente, a busca pela lista inteira, a menos que também existe uma ordenação. Maior vantagem: mais expansível.

No entanto, ainda assim, a alocação contígua gera um grande problema, a fragmentação de espaços livres, o qual, apesar de haver espaço livre no disco, o não existe um segmento contíguo onde o arquivo pode ser alocado. Para resolver este

problema deve ser feita a defragmentação periodicamente para reorganizar os arquivos no disco, a fim de que exista um único segmento de blocos livres. No entanto, esta solução tem efeito temporário e gasta bastante tempo.

Para compensar a necessidade de ter de existir um único bloco contíguo contendo o espaço necessário, a abordagem de alocação encadeada seria uma bela solução, visto que o arquivo é organizado como um conjunto de blocos ligados no disco, independentemente de sua localização, e cada bloco aponta para o próximo bloco. Para isto ser possível, é necessário fragmentar os arquivos em blocos (chamados de *extents*). No entanto, como tudo não é belo, este tipo de abordagem aumenta o tempo de acesso ao arquivo, pois o braço do disco (algo físico) necessita mover mais vezes para acessar todas as *extents*. Além disso, temos mais três desvantagens que é a defragmentação frequente do disco, a alocação só permite acesso sequencial e desperdiça espaço nos blocos para o armazenamento dos ponteiros.

Para resolver o problema manutenção de ponteiros, se criou a lista indexada, cujo principal princípio é manter um único bloco que contém todos os ponteiros para os blocos do arquivo, chamado de bloco de índice. Isto permite um acesso direto aos blocos do arquivo (visto que não estão conectados) e não se faz necessário ter informações de controle dentro dos blocos de dados. Devido ao grande armazenamento dos discos atuais, não se considera uma perda grande ter um bloco que armazena ponteiros para todos os blocos do arquivo.

Fragmentação Interna

Enquanto que o a fragmentação externa são os espaços vazios entre os blocos de arquivos no disco, a fragmentação interna é a transformação do arquivo em blocos. Normalmente são utilizados blocos de tamanho fixo entre 512 byte até 8kbytes, sendo que blocos não podem ser alocados parcialmente. No entanto, ainda assim, existe a chance desperdício de memória e para se perceber o porquê deste desperdício, imaginemos que usemos blocos de 4096 bytes, um arquivo de 5700 bytes ocupará 2 blocos e o segundo bloco perderá 2492 bytes, visto que não poderá ser alocado para outro arquivo. Em média, temos o desperdício de meio bloco por arquivo.

Logo, temos um grande problema, qual é o tamanho ideal dos blocos? Bem, a resposta não é simples de responder, mas fica a depender do que se deseja e do que se tem disponível, pois:

- Em blocos pequenos: menor perda de fragmentação interna, pois a chance de desperdiçar espaço é muito menor, já aumentamos a necessidade de um melhor gerenciamento destes blocos pelo aumento da quantidade.
- Em blocos grandes: maior perda de fragmentação interna, pois é muito provável que um arquivo não ocupe todo o bloco, no entanto, ganhamos com a gerência, pois temos bem menos blocos para tomar conta.

Proteção de acesso

Bem, além de tudo que já estudamos para um melhor gerenciamento de arquivo, também temos de levar em conta usuários que tem acesso aos arquivos, pois, dado que mais de um usuário têm acesso a um certo arquivo, temos de tomar

cuidado no gerenciamento deste arquivo de modo evitar conflitos de acesso. Devido a isso, cada sistema de gerenciamento deve conter seu mecanismo de proteção de acesso, dando concessão de leitura, gravação, execução e/ou eliminação, a depender do nível de acesso.

Para realizar tal procedimento, o S.O. separa os usuários em grupos, de modo a compartilhar arquivos entre si, gerando então três níveis de proteção: proprietário, grupo e todos. Na criação de um arquivo, a concessão dos três níveis são definidos, ficando a modificação do arquivo a depender do dono e usuários privilegiados.

A manutenção das propriedades de proteção de acesso ficam localizadas numa lista chamada *Acess Control List*. O tamanho da lista pode ser bastante extensa, caso o compartilhamento seja com uma grande quantidade de usuários. Devido a isso, existe um *overhead* adicional para que haja a pesquisa, realizada de modo sequencial, que o sistema realizará sempre que for solicitado.

Implementação de Caches

Devido ao acesso ao disco ser bastante lento, se comparado a outras formas de acesso a arquivos, este é considerado o fator para que as operações de entrada e saída sejam consideradas um problema para o desempenho de sistema. De modo a minimizar este problema, foi implementado uma técnica chamada de *buffer cache*, onde o sistema reserva uma área na memória para que se tornem disponíveis e utilizadas em operações de acesso a disco.

Quando o sistema procura por um arquivo, ele é primeiro procura na *cache* e, caso não encontrado, é procurado na memória em disco e, posteriormente, atualiza o *buffer cache*.

Além disso, assim como na memória, o *cache* possui espaço limitado e por isso adota políticas de substituição assim como na memória. Concomitantemente, como os dados podem ficar muito tempo na *cache*, a ocorrência de perdas de energia e outros fatores podem enganar o sistema de modo a fazer ele acreditar que já tenha salvo em disco o bloco modificado, quando na verdade não foi feito. Para que isto não ocorra, existem duas maneiras para manter os dados em disco atualizados:

- Rotina: o sistema possui uma rotina que, de tempos em tempos, atualiza o disco com todas as modificações feitas na *cache*.
- *Write-through caches*: toda vez que um bloco da *cache* for modificado, se realiza uma atualização no disco.

Princípios de Hardware para entrada e saída

Como o sistema de entrada e saída é uma combinação de software e hardware, temos de estudar melhor como o hardware opera por sobre as operações de E/S. Além da diversidade de dispositivos (inumerável aqui), também existem características que são relevantes para cada tipo, vejamos algumas delas:

- Com relação a transferência
  - Por caracter: transferem bytes um a um – terminal.
  - Por bloco: transferem bytes em blocos – disco.
- Ordem de acesso
  - Sequencial: acesso em ordem fixa – modem.
  - Acesso randômico: ordem pode ser alterada – CD-ROM.
- Tempo de resposta:

    - Síncrono: tempo de resposta previsível – fita.
    - Assíncrono: tempo de resposta imprevisível – teclado.
- Nível de compartilhamento
    - Compartilhável: podem ser acessado/usados por vários processos ao mesmo tempo – teclado.
    - Dedicado: apenas um único processo pode utilizá-lo por vez – impressora.
- Nível de proteção de acesso:
    - Leitura e escrita
    - Somente leitura
    - Somente escrita

Arquitetura de E/S de Hardware

Para que dispositivos periféricos sejam acessados pelo sistema central, existem os *device Drivers* que são softwares que fornecem uma interface de acesso uniforme ao dispositivo capazes de traduzir as chamadas do usuário para o dispositivo específico, convertendo dados e detectando e corrigindo erros específicos. Mas então fica a questão, como a CPU acessa a informação destes dispositivos?

Através de espaços de endereçamento – conjunto de endereços de memória que o processador consegue acessar diretamente – ou de alguma forma definida no projeto do processador, podendo haver três possibilidades:

- E/S isolada: utiliza instruções especiais de E/S, especificando a leitura/escrita de dados numa porta de E/S. Deste modo temos dois endereços, um na memória principal, outra da porta do dispositivo de E/S.
- E/S mapeado na memória: utiliza instruções de leitura/escrita na memória, pois os dados do dispositivo ficam guardados no espaço de endereçamento, havendo então apenas um endereço (o da memória). Neste caso, podemos ter uma arquitetura de barramento único (uma única entrada para na CPU para memória e E/S) ou de barramento duplo (memória utiliza um barramento exclusivo e de alta velocidade, enquanto que dispositivos de E/S utilizam o barramento geral).
- Híbrido: une-se ambos os mecanismos, havendo então dois espaços de endereçamento, um na memória e outro na porta do dispositivo.

Tendo acesso a informações, é importante saber como o processador se comunica com os dispositivos externos. No caso de acesso a memória principal, o processador tem acesso direto e sem intermediários, no entanto, no caso de dispositivos externos, a CPU utiliza de circuitos intermediários que são chamados de interfaces. Deste modo, quando o processador deseja fazer operações de leitura ou escrita, este fazer estas ações na interface do dispositivo.

Além disso, existe também um dispositivo de hardware chamado DMA (*Direct Acess Memory*), que fica responsável por transferir dados do dispositivo externo para a memória (ou vice-versa) e passar confirmações de transferência de volta a CPU.

Sabemos então que foi pedido para fazer uma ação num dispositivo de E/S, a questão que fica é, quando sabemos que acabou o processamento no dispositivo? Bem, existem três tipos mecanismo para tomar conhecimento da finalização:

- E/S Programada: CPU fica constantemente lendo o status do controlador e verificando se o processo de E/S já acabou (*Polling* ou *Busy-waiting*). Sua maior desvantagem é a espera ociosa pela finalização do procedimento.
- E/S por Interrupção: CPU é interrompido ao fim do procedimento de E/S pelo dispositivo e então transfere os dados. Isto permite que a CPU continue executando enquanto que o procedimento acontece no dispositivo. No entanto, ainda assim não é muito ideal, pois todo procedimento para dispositivo de E/S precisa passar pela CPU.
- E/S por DMA: Quando necessário, o controlador de E/S solicita ao controlador de DMA a transferência de dados de/para a memória, não havendo envolvimento da CPU nas fases de transferência. Apenas ao término, a CPU é informado que foi finalizado o procedimento.

Princípios de Software de E/S

Como todo e qualquer função de interfaces de software de sistemas, seus objetivos são busca pela eficiência, uniformidade na hora de acesso a dispositivos, transparência de detalhes para as camadas mais superiores e oferecer abstrações genéricas para os dispositivos, facilitando o uso por outros programas.

No entanto, nem nada é tão simples assim. Devido à alta diversidade dos periféricos, o sistema de gerenciamento de E/S é bastante complexo, pois temos de padronizar ao máximo para reduzir número de rotinas e fazer com que novos dispositivos não alterem a visão do usuário com relação ao sistema geral. Por este motivo, foi criado uma camada responsável pelo gerenciamento desta diversidade de dispositivos.

Camadas de Software de E/S

Como cada dispositivo externo de hardware é único, seus drivers permitem que o S.O. tenha melhor abstração em como acessar esses dispositivos. No entanto, isto não é suficiente para o usuário. O que este precisa é de uma interface independente do dispositivo de modo que este posso fazer uma chamada de E/S sem se preocupar com qual tipo de dispositivo este está lidando. Deste modo, temos 4 camadas (e a mais abaixo o hardware), responsável por esta troca de dados entre usuário e dispositivo:

- Software de E/S no nível do usuário: responsável pela chamada de E/S do usuário, colocando este no *spool*.
- Software de S.O. independente do dispositivo: oferece bufferização, sistemas de nomes para os arquivos do dispositivo – assim como ocorre nos arquivos do próprio computador, e proteção de acesso aos reservados usuários. Além disso, a alocação e liberação dos recursos fica responsável pelo S.O. Pode ser considerado como uma interface uniforme para os drivers do dispositivo.
- Drivers do dispositivo: Recebe aquisições do S.O. e configura o controlador do dispositivo. Este oferece uma posição lógica para os dispositivos, para serem buscados pelo S.O. A comunicação entre drivers e controladores é feita através do barramento.

- Tratador de Interrupções: acionado ao final da operação de transferência de dados e aciona o driver. As interrupções devem ser transparentes o máximo possível e isto é feito através do bloqueamento do driver que iniciou a operação de E/S até que uma interrupção notifique que o procedimento terminou e desbloqueie o driver. Isto impede que a CPU realize estas operações que não fazem parte do seu repertório de instruções.

Sistemas Distribuídos e Infraestrutura de suporte

O que lidamos no dia-a-dia, em nossos computadores, é um sistema de computação centralizado, onde uma única máquina fica responsável pelo processamento de processos. No entanto, a computação avançou e foi-nos permitido interpretar e processar instruções em mais de um processador (CPU), concorrentemente ou paralelamente. Tal "novo" sistema é conhecido como sistema distribuído. Podemos definir sistemas distribuídos como um conjunto de processos autônomos que comunicam-se entre si.

Podemos melhor definir um sistema distribuído como um sistema de aparenta ser centralizado do ponto de vista do usuário, mas capaz de executar tarefas em múltiplas CPUs independentes. O conceito chave é transparência, pois o uso de múltiplos processadores se torna invisível ao usuário, criando a ilusão de um único processador virtual e não uma coleção de máquinas.

Um resumo de várias definições é que sistema distribuído é um conjunto de máquinas independentes com seu vários componentes de software e hardware, conectados de alguma maneiras – seja rede ou barramentos -, compartilhando recursos com o objetivo de dividir uma tarefa de modo transparente ao usuário.

Por que Distribuir?

Distribuir o sistema não parece ser uma tarefa fácil, no entanto, como tudo, existe uma razão de porque estamos buscando fazer isto. Vamos entender então cada um destes motivos:

- Manutenção – dividir para conquistar: é o lema do próprio lema, e basicamente consiste em dividir uma certa tarefa em tarefas menores e autônomas capazes de se comunicar. Isto, além de facilitar o código, agiliza a manutenção do sistema devido a modularidade.
- Acomodação ás capacidades das máquinas disponíveis: de vez em quando, um programa que não consegue rodar em uma única máquina, ao ser distribuído, poderá efetivamente funcionar.
- Melhor desempenho e Escalabilidade: isso advém do fato que temos um paralelismo real ou concorrência ocorrendo.
- Tolerância a falhas: como o sistema é distribuído, caso uma das máquinas falhem, por exemplo, se o programa está dentro de outras máquinas, o usuário não sentirá que ocorreu uma falha.

A grande utilização de sistemas distribuídos é seu potencial superior ao de um sistema centralizado, pois, além de poder ser mais confiável visto que toda função

pode ser replicada – de modo que se uma máquina falha ou um disco é perdido, existirá outra máquina ou outro disco para substituir o problema – ainda há a computação paralela, trazendo assim uma maior vazão de processos. Com isso, podemos considerar como as propriedade fundamentais dos sistemas distribuídos a tolerância à falha e o paralelismo.

Além das razões de desempenho, construir sistemas distribuídos nos fornece um ambiente capaz de ser expandido incrementalmente, com disponibilidade de recursos – graças a sua capacidade de replicação e redundância – e uma grande modularidade dos componentes e serviços oferecidos. Concomitantemente, o fator escalabilidade é grande nos sistemas distribuídos, visto que não existe qualquer componente central que limita o tamanho do sistema, assim como o fator de confiança citado anteriormente, já que além da disponibilidade dos serviços e componentes, ainda há a recuperação das falhas.

Complexidade e Problemas de Sistemas

Bem, apesar de termos falados de todas as vantagens dos sistemas distribuídos, temos que deixar claro que é necessário entender sua complexidade e as falhas que podem surgir com sua implementação. É devido à complexidade que se limita o poder de construção de um sistema e isso nos trás alguns problemas, chamados de problemas de sistemas. Vamos analisar os tipos de problemas de sistemas que temos em sistema distribuídos:

- Interconexão: os problemas que surgem ao se conectar componentes que antes eram independentes.
- Interferência: os problemas que surgem quando dois componentes que antes tinham um comportamento razoável quando separados, exibem um distúrbio quando combinados.
- Propagação de efeito: um problema de falha com o poder de derrubar um sistema inteiro devido ao seu poder de propagação entre os componentes do sistema.
- Efeitos de escala: um sistema que funciona bem com um certo número de nós, mas, ao ser expandido, tem seu funcionamento desviado do desejado.
- Falha parcial: ocorre quando um componente do sistema distribuído falha, mas não o sistema como um todo. A essência dos sistemas distribuídos e a fonte considerável de complexidade em aplicações tolerantes a falhas.

Os sistemas distribuídos são complexos, porque suas tarefas são complexas, há uma necessidade de gerenciar tanto as falhas e a disponibilidade, como os próprios sistemas de arquivos, a concorrência, entre outros fatores. E, além disso, há uma necessidade de conciliar o custo com o benefício de modo que não seja criado nada muito custoso, no entanto, sem benefício ou muito complexo para ser implementado.

Transparência

Sabemos que sistemas distribuídos nos trazem grandes vantagens como distribuição do problema, comunicação entre os componentes, complexidade relativa e uma heterogeneidade para suportar os diferentes componentes do sistema. No entanto, nada disto seria possível sem uma transparência que possibilitasse a fácil

implementação dos mesmo. Por conta disso, vamos dar uma pincelada nos tipos de transparências para ter melhor ideia de como os sistemas distribuídos trabalham:

- Transparência de localização: o usuário não sabe onde o recurso está, apesar de ter acesso ao mesmo sem grandes problemas.
- Transparência de acesso: operações seguem um padrão para um acesso local ou remoto, não dificultando o método de acesso.
- Transparência de migração: o recurso é capaz de se mover para outra localidade sem causar transtorno ao acesso do mesmo.
- Transparência de relocação: o recurso é capaz de ser transferido para outra localização, mesmo quando em uso.
- Transparência de concorrência: os recursos são compartilhados sem interferir em outros processos.
- Transparência de falha: quando um recurso falha e é recuperado, o usuário não é capaz de notar o processo de falha e recuperação.
- Transparência de replicação: os usuário do sistema não tomam ciência da existência de recursos duplicados.

Infraestrutura de sistemas distribuídos

Entender como os sistemas distribuídos se organizam nos faz melhor entender como seu comportamento e serviços são oferecidos para o usuário. Por esta razão, vamos dar uma breve estudada na organização dos sistemas distribuídos e mostrar uma comparação com outros sistemas.

Sistemas distribuídos são como sistemas operacionais tradicionais, são responsáveis pelo gerenciamento dos componentes do sistema e oferecem uma abstração da complexidade de baixo nível, gerando uma melhora na interação com o usuário. Os sistemas distribuídos vão além disso e passam também a gerar uma transparência da heterogeneidade dos componentes e da complexidade de hardware, através de uma máquina virtual onde as aplicações podem ser facilmente executadas.

Com isso, podemos ter dois tipos de sistemas operacionais para sistemas distribuídos: aqueles que mantém uma visão global e única dos recursos gerenciados – fortemente acoplados – e aqueles que cada máquina tem seu próprio gerenciamento de recursos e que cooperação entre si, oferecendo seus recursos disponíveis as outras máquinas – fracamente acoplados.

Então nos surge dois sistemas operacionais: os Sistemas operacionais distribuídos, que são fortemente acoplados, e os sistemas operacionais de rede, que são fracamente acoplados e suas aplicações estão distribuídos nas máquinas. Em ambos os casos é necessário um middleware para ser permitido uma maior escalabilidade e abertura nos sistemas de rede, enquanto que gera maior transparência e uso mais fácil dos sistemas distribuídos.

Sistemas Operacionais Distribuídos

Os SODs são essencialmente sistemas operacionais tradicionais, exceto que manipulam múltiplas CPUs, sejam elas dentro da mesma máquina – multi-processador – sejam elas em máquinas diferentes – multi-computador.

O SO multi-processador visa ter um alto desempenho através de múltiplos processadores, sendo o uso da CPU transparente para a aplicação. Além disso, há uma memória compartilhada com proteção de acesso para garantir consistência de

dados através de primitivas de sincronização – já estudadas anteriormente em questões de threads e processos. Já em um sistema de multi-computador, a palavra-chave é comunicação, pois os serviços das aplicações e o próprio sistema operacional está distribuído nas máquinas, que se comunicam para obter os serviços desejados. Para isso é utilizado um middleware.

Middleware

Enquanto que temos em um extremo os sistemas operacionais distribuídos que utilizam sistemas homogêneos, com transparência de distribuição, alto desempenho, espaço de memória compartilhada e um alto controle de concorrência, temos no outro extremo os sistemas operacionais de rede que utilizam sistemas heterogêneos com pouca transparência, alta escalabilidade e comunicação intensa.

No entanto, existe um médio termo entre esses dois tipos de sistemas que acaba oferecendo complementos que faltam em ambos os extremos. O middleware entra como um mecanismo que tem preferência para sistemas heterogêneos, com capacidade de transparência de distribuição e comunicação, uma gama de serviços e abertura para escalação. Ou seja, o middleware é uma camada entre as aplicações e o sistema operacional que permite uma integração dos sistemas heterogêneos, com uma interoperabilidade entre eles e maior portabilidade dos serviços através de uma comunicação transparente.

Uma definição mais formal de middleware é um conjunto reusável e expansível de serviços e funções que são comumente necessários por parte de várias aplicações distribuídas para serem capaz de funcionar em um ambiente de rede.

Alguns deste serviços comumente oferecidos pelo middleware são:

- Facilidade de comunicação de alto nível: implementação de uma transparência de acesso a garantir uma maior facilidade na passagem de mensagem na rede.
- Serviço de nomes: implementa a transparência de localização.
- Persistência: possui um sistema de arquivos distribuídos que garante a persistência.
- Transações distribuídas e segurança.

De modo a garantir a propriedade de portabilidade, o middleware cria uma API de modo a firmar uma fácil utilização do mesmo pelas aplicações, constituído por funcionalidade comuns entre elas – visão Top-down. Além disso, são necessários protocolos de troca de mensagem para que haja interoperabilidade entre os sistemas heterogêneos, escondendo problemas complexos que não são de interesse das aplicações – visão Bottom-up.

Serviço de Nomeação

Um dos mais importantes serviços de um middleware e o responsável pela transparência de localização dos recursos e serviços. Para isso é necessário a transformação dos endereços físicos das máquinas para nomes convenientes a olhos humanos. Surge a necessidade de mapeamento do endereço físico ao nome, sendo o responsável por este mapeamento o servidor de nomes, um agente intermediário do middleware.

Além de oferecer uma transparência de localização para a aplicação, o serviço de nomes também tem a habilidade de ser escalar, ter um longo tempo de vida, ser altamente disponível, ter isolamento de falhas e tolerar a desconfiança.

Existem três tipos de navegação para obter os dados de nomes de modo a obter o mapeamento completo de um nome nos diferentes servidores distribuídos:

- Navegação Interativo/Multicast: O cliente fica responsável por todo o processos de navegação, é ele que vai atrás dos servidores para obter o mapeamento.
- Navegação não-recursiva controlada por servidor: O cliente nomeia um servidor de nomes para realizar o processos de navegação por ele e este utiliza uma navegação interativa para obter o mapeamento.
- Navegação recursiva controlada por servidor: A tarefa é passada de servidor a servidor de modo que, ao encontrar o servidor que sabe o mapeamento, a recursão é desfeita.

Serviço de Comunicação

Além do serviço de nomes para gerar a transparência de localidade, o middleware também oferece um serviço de comunicação de modo a garantir uma transparência na comunicação e elevar o nível de comunicação entre os sistemas. Imaginemos uma transmissão de dados ocorrendo entre dois sistemas heterogêneos, de modo que os dados em cada programa dos sistemas são estruturados unicamente para o sistema que o gerou. Então existe um heterogeneidade na geração dos dados entre os sistemas, é necessário garantir a utilização de um formato único para o uso externo comum entre eles e a capacidade de retornar ao formato original.

Para alcançar tais objetivos foram propostos dois mecanismo:

- Marshalling: Formalmente definido com uma linearização dos dados estruturados para cada sistema. Mas nada mais é do que uma tradução de dados para um formato externo único - XDR (*–xternal Data Representation*).
- Unmarshalling: Tradução do formato externo para o formato local através de um segmento enviado que define para que estrutura os dados devem ser restaurados.

Para facilitar ainda mais a comunicação, foi desenvolvido para o middleware um mecanismo chamado de RPC (*Remote Procedure Call*) que tem como objetivo criar um ambiente que seja capaz de programar um sistema distribuído como se fosse um local centralizado, através de chamadas de procedimentos remotos como se fossem locais, ocultando a entrada e saída de mensagens. Além disso, o RPC evita a passagem de endereço e as variáveis globais para a troca de dados.

Para possibilitar uma melhor integração do mecanismo de RPC com o seu objetivo de um acesso remoto feito através de uma chamada local, é necessário criar uma interface capaz de fazer a conversão. Isto é feito através de um Stub em ambas as partes, que garante uma transparência de acesso para o RPC não ter de se preocupar com a localização do recurso, podendo realizar uma chamada remota como uma chamada local. Lembrando que é necessário fazer alguns tratamentos de exceções no local, além da utilizam de marshalling e unmarshalling para a troca de informação.

Um Stub funciona com a mesma dinâmica de um cliente-servidor e cada um tem sua função distinta:

- Funções do cliente:
  - Intercepta a chamada e empacota os dados (marshalling), enviado então a mensagem para o servidor através do núcleo.
  - Desempacote a resposta recebido pelo núcleo e passa o resultado para a chamada.
- Funções do servidor:
  - Recebe a chamada através do núcleo, desempacotando os dados (unmarshalling) e chama o procedimento local responsável pela execução.
  - Empacota os resultados e envia a resposta para o cliente que o convocou através do núcleo.