**Questão 04) Resumo do vídeo sobre Linux**

**Tópicos :**

Esse tópico introduz de forma resumida a história do Linux e o porquê que ele foi criado.

Lixux foi criado por Linus Torval em 1991, que não estava contente com o SO Unix e decididiu programar seu próprio SO baseado em Unix, porém mais potente e com suas próprias particularidades. Hoje em dia o Linux é um dos SO’s mais repandidos no mundo e é usado em diversas máquinas desde pequenos objetos como smartwatchs até grandes aparelhos como os PC’s.

Trata sobre as semelhanças e diferenças do Linux com outros SO’s.

Os kernels são divididos em dois tipos de design:

1. Monolitico: Unico processo rodando em um único espaço de endereço, existem no disco como *single static binaries* e tudo que é rodado nesse kernel é rodado nesse único espaço.
2. MicroKernel: A funcionalidade desse kernel é dividida em vários processos, usualmente chamados de servidores separados em diferentes endereços de espaço. Nesse kernel, a comunicação é dada por troca de mensagens no IPC (*Inter-process communication*) entre os processos. Esse separação previne que caso um processo parar de funcionar, somente ele pára e não todo o kernel, visto que ele está separado dos outros processos, diferente do Monolitico.

Windows XP e 7 são MicroKernels enquanto que o Linux é Monolitico

Linux suporta carregamento dinâmico de módulos do kernel, em comparação com o Unix.

Linux é *open-source*.

**05)**

Trata da diferença da programação no Espaço de Usuário e no Espaço do Kernel.

A memória virtual é dividida entre os dois, o espaço de kernel é a área onde os processos do kernel rodam e o espaço do usuário é onde os processos de usuário rodam. Esse divisão serve para proteger o acesso à memória.

O kernel é programado em GNU C, usando várias bibliotecas da linguagem C.

A proteção da memória do kernel é menor do que da memória do espaço de usuário.

A pilha de memória do Kernel tem tamanho fixo e não mutável, diferente da de usuário, que pode ser alocada dinâmicamente. Nos sistemas x86, a pilha pode ter tamanho de 16KB e nos de 32bits ela tem 8KB.

O kernel pode ser preemptivo, ou seja, o kernel pode matar qualquer processos à qualquer momento caso este esteja causando algum problema prevenindo assim problemas com outros processos.

**06)**

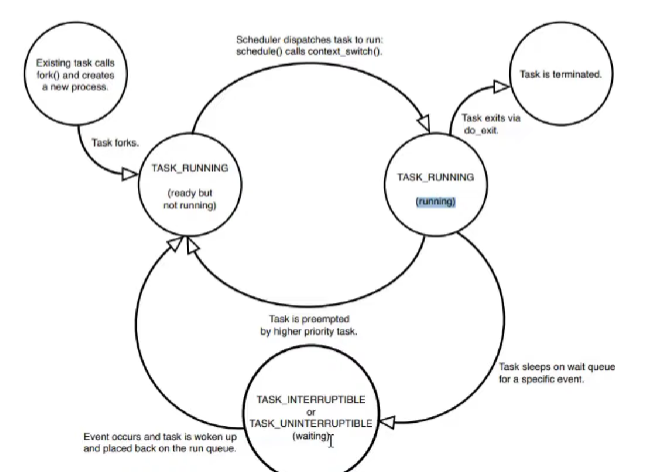
Trata de como os processos são restreados e controlados no kernel.

Todo programa que está num estado de execução é chamado de processo. Nem todo programa é um processos, apenas aqueles que requerem algum recurso da memória.

Um processo é criado com uma chamada *fork()* que cria um novo processo duplicando um já existente, o processo que chama o *fork()* é o parente e o novo processo é o filho.

No linux cada processo tem um PID (*Unique process identification*), esse número é então usado para gerir o processo, podemos pará-lo, matá-lo etc utilizando da sua identificação.

O controle de processos funciona como seguinte:



**07)**

Explica o que são threads e como elas são tratadas no Kernel do Linux.

Uma thread é a menor unidade de processamento que pode ser performada em um SO. Threads estão contidas nos processos e cada processo pode conter múltiplas threads. As threads permitem que processos rodem simultaneamente, não sequencialmente.

Thread permite o que conhecemos por Programação Paralela.

Kernel Threads não possuem endereço de espaço e o Linux não considera as threads como sendo threads normais, ele implementa suas threads como sendo processos.

**08)**

Mostra como os processos de kernel são agendados (*scheduling*) e quais algoritmos de agendamento de processos existem.

O Linux possui um sistema de gerenciamento separado para tratar os processos, o *process scheduler*, o qual divide os recursos finito do tempo de processo entre os processos em execução num sistema, evitando qualquer caso de conflito entre eles ou casos de gargalo.

O *scheduler* atribui os recursos para cada processo de acordo com as suas necessidades. Exemplo, Ele atribui recursos a um processo, depois o mesmo dorme, dando espaço a outro processo a ser executado, assim dando a impressão de que os processos estão rodando simultaneamente.

Para saber em qual ordem executar os processos o *scheduler* adota uma política de gerenciamento:

- A primeira que temos é a I/O-Bound vs Processor-Bound processes, onde o Linux dá mais prioridade aos I/O-Bound processes.

- Prioridade: Alguns processos tem prioridade maior que outros.

A prioridade pode ser dada pelo:

1. *nice value* que varia de um valor de -20 a 19, onde quanto menor o valor menor a prioridade.
2. *Real-time priority*, que varia de 0 a 99. Todos os processos de tempo real tem mais prioridade que os processos normais
3. *Timeslice*, valor que representa quanto tempo o processo pode rodar. Quanto maior o tempo pior será a performance interativa do sistema

**09)**

Explica o que é uma chamada de sistema e como usá-la.

Uma chamada de sistema é um meio de permitir a comunicação de processos rodando no espaço de usuário podem se comunicar com o espaço do sistema (Comunicação Hardware-User Space).

Tipicamente acessados pela função *call*. No Linux, para cada chamada *call* é atribuído um número.

Quando um programa chama um servico do kernel ele não chama diretamente esse serviço, isso não é permitido no Linux, em contrapartida, o Linux tem o chamado *System Call Handle* que vai tratar essa comunicação e ativar o modo kernel quando necessário, realizando assim a função de um intermediário entre o programa e o serviço.

**10)**

Esse tópico ensina como implementar uma chamada de sistema no kernel.

Uma chamada desistema deve ter uma interface limpa e simples com o menor número de argumentos possível. O seu objetivo vai se manter constante, mas o seu uso pode mudar.

A chamada de sistema vai verificar todos os parâmetros passados para impedir que parâmetros não válidos e ilegais perigosos ao sistema passem.

**12 e 13)**

Define o que é uma interrupção e como ela é tratado no kernel.

Uma interrupção é uma resposta de um processo a um evento que precisa de atenção do software, como por exemplo apertar uma tecla no teclado enquanto um programa roda.

A interrupção possibilita o hardware de enviar um sinal ao processador, que por sua vez ao SO de responder a esse novo dado.

Dispositivos diferentes são associados com diferentes interrupções através de valores únicos associados às interrupções, por exemplo, as interrupções do disco rígido são diferentes das do teclado.

Dessa forma, cada interrupção pode ser gerida pelo seu próprio gerenciador. Esses valores são usalmente chamados de IRQ (*interrupt requests*). Por exemplo, no pc IRQ0 é a interrupção *timer* e o IRQ1 é o do teclado.

O gereciador de interrupções (*interrupt handler*) é dividido em dois:

- *Top Halves*: é rodado imediatamente e perfoma trabalhos com tempo crítico.

- *Bottom Halves*: Trata trabalhos que não tem tempo crítico e podem ser feitos depois

**15)**

Trata das regiões críticas (*critical region*)e condições de corrida (*race conditions*) e como protegê-las.

Uma região crítica é um caminho do código (PATH) que acessa e manipula os dados compartilhados da aplicação, aqui, apenas um código por vez é executado, e se tem mais de um programa tentando ser executado nessa região então temos uma condição de corrida para conseguir os recursos.

Uma condição de corrida é uma condição indesejada que ocorre quando um dispositivo ou sistema tenta realizar duas ou mais operações ao mesmo tempo.

De forma a prevenir essa corrida existem diferetes técnicas de bloqueio (*locking techniques*) que trava a região crítica quando um processo está sendo executado nela para evitar que outro processo comece sua execução, então, quando o primeiro processo terminar, o segundo entra na região crítica e é executado, travando a mesma.

**17)**

Mostra como o kernel gere o tempo e como ele mantém o tempo atualizado num sistema.

Existem funções no kernel dependentes do tempo e dependentes de eventos, sendo que a maioria das funções são dependentes do tempo, então por isso que é importante que o kernel trate da passagem do tempo.

O Kernel trabalha então com o hardware, o qual provém um timer do sistema que o kernel usa para medir a passagem de tempo.

O timer pode funcionar usando do tempo de *clock* do processador, a sua frequência, a cada vez que um período, o timer aciona um *tick*, então o kernel conhecendo o tempo entre dois *ticks* consegue medir a passagem do tempo.

O kernel também consegue recuperar o valor do tempo do dia (h-min-sec ...) que é útil para o espaço de usuário.

**19)**

Esse tópico trata de como o kernel gere a memória e como o kernel lida com a memória pequena que lhe é atribuída.

O kernel tem memória bastante limitada, então ele tem que evitar gargalos. É mais fácil gerir a memória no espaço de usuário, onde cada processo recebe o tamanho de memória necessário, mas no kernel é mais difícil alocar memória.

O tópico trata também de *paging*, onde um computador pega dados das memórias secundárias e os usam na memória principal. Nesse esquema, o sistema operacional recupera dados do armazenamento secundário em blocos de mesmo tamanho chamados *pages*. Isso é útilm pois usa a memória secundária para permitir que programas excedam o tamanho disponível na memória física.

A páginas recebem valores para indicar se ela está sendo usada ou não, se o valor for negativo ou 0 a página não está sendo usada, se for maior que será, ela está sendo usada.

Outro modo de gerência de memória do kernel inclui “zonas”, que é um mode de juntar as páginas semelhantes, visto que nem todas as páginas podem ser usadas pelo kernel e elas possuem objetivos diferentes

**24)**

Trata sobre a Camada de Abstração do sistema de arquivos.

O kernel do Linux implementa uma camada de abstração entorno da sua interface do sistema de arquivos de baixo nível, o que permite que uma interface genérica para todo tipo de sistema de arquivo seja possível.

Uma camada de abstração é uma forma de esconder os detalhes de trabalho de um subsistema, permitindo a separação de preocupações para facilitar a interoperabilidade e independencia de plataformas.

Essa camada permite ao Linux de suportar diferentes sistemas de arquivos, o que é possível porque o VFS (sistema de arquivos virtual) provém um modelo comum de arquivo que representa qualquer característica geral e comportamento de um sistema de arquivos