



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA
GRADUAÇÃO EM ENGENHARIA MECATRÔNICA



Tarefa Semanal 03 – Sistemas Embarcados II

Linux como ambiente de Programação

Fernando Rabelo Fernandes Junior - 11611EMT020

Uberlândia, janeiro de 2021

Resumo Capitulo 3 - Advanced Linux Programming

3. Processos

Se você tem duas janelas do terminal exibidas na tela, provavelmente você está executando o mesmo programa de terminal duas vezes. Em cada terminal, a janela provavelmente está executando um shell, cada shell em execução é outro processo. Compilar um comando de um shell, faz com que o programa correspondente seja executado em um novo processo. o processo do shell é retomado quando esse processo é concluído. Programadores avançados costumam usar vários processos cooperativos em um único aplicativo para permitir que o aplicativo faça mais de uma coisa ao mesmo tempo, para aumentar robustez da aplicação e fazer uso de programas já existentes.

3.1. Olhando para os processos

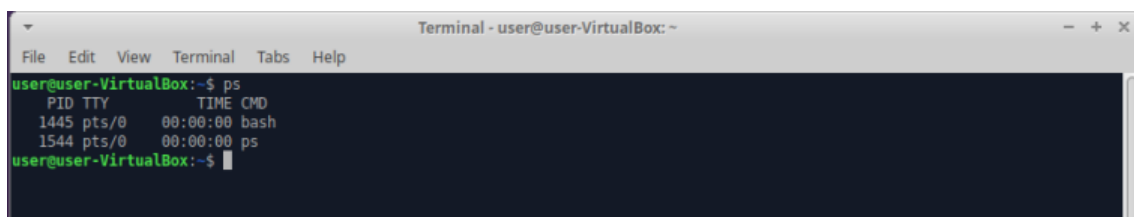
Mesmo quando você está sentado em frente ao seu computador, há processos em execução. Cada execução programa usa um ou mais processos.

3.1.1. Processos Ids

Cada processo em um sistema Linux é identificado por seu ID de processo exclusivo, às vezes conhecido como pid. IDs de processo são números de 16 bits que são atribuídos sequencialmente por Linux à medida que novos processos são criados. Um programa pode obter o ID do processo de o processo em que está sendo executado com a chamada de sistema getpid() e pode obter o processo ID de seu processo pai com a chamada de sistema getppid().

3.2.2. Visualizando Processos Ativos

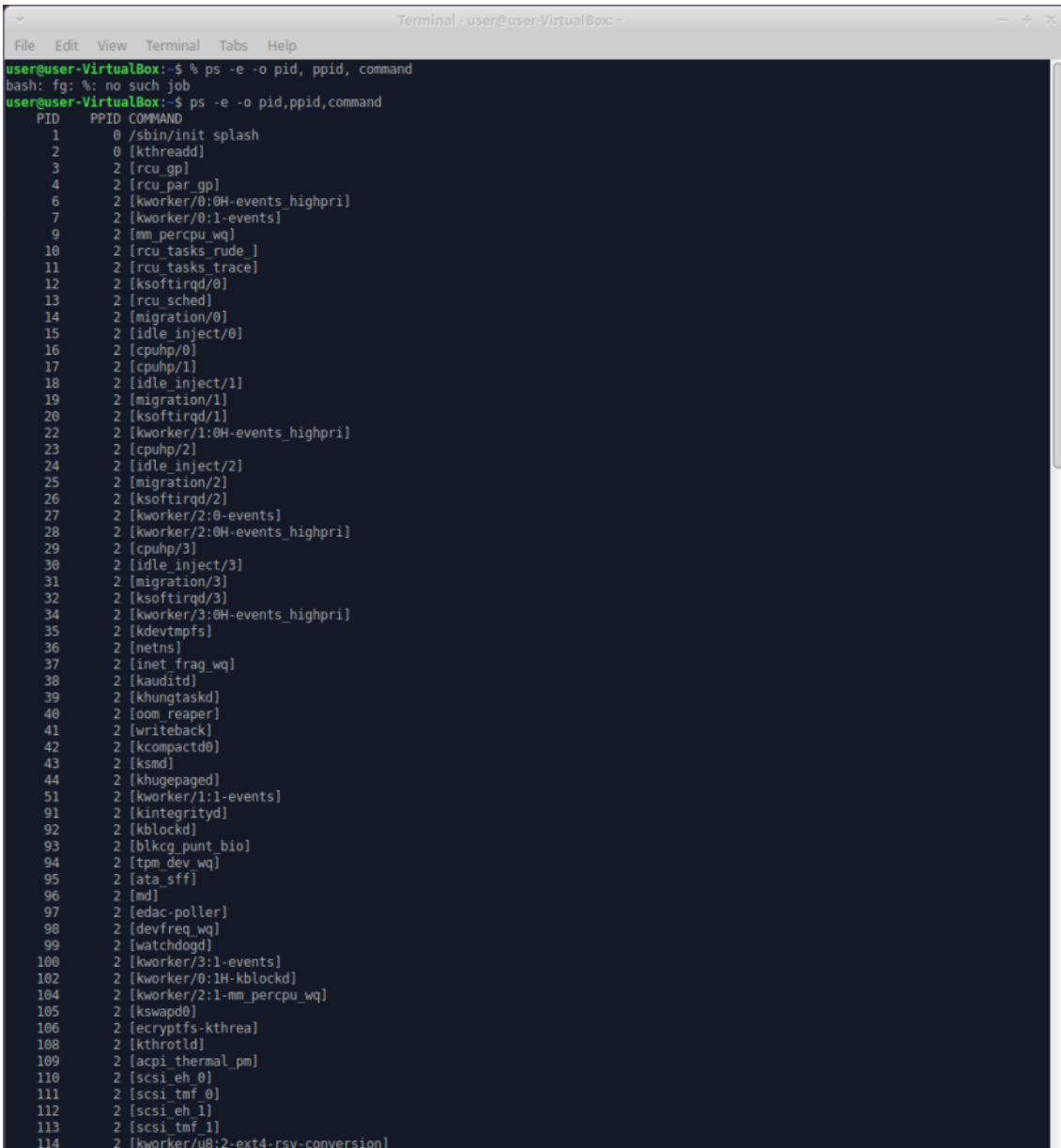
O comando ps exibe os processos que estão sendo executados em seu sistema. A versão GNU/Linux do ps tem muitas opções porque tenta ser compatível com versões do ps em várias outras variantes do UNIX.



```
Terminal - user@user-VirtualBox: ~
File Edit View Terminal Tabs Help
user@user-VirtualBox:~$ ps
  PID TTY          TIME CMD
 1445 pts/0    00:00:00 bash
 1544 pts/0    00:00:00 ps
user@user-VirtualBox:~$
```

Esta invocação de ps mostra dois processos. O primeiro, bash, é o shell rodando neste

terminal. A segunda é a instância em execução do próprio programa ps. A primeira coluna, rotulada PID, exibe o ID do processo de cada um. Para uma visão mais detalhada do que está sendo executado em seu sistema GNU/Linux, invoque isto:



```
Terminal - user@user-VirtualBox:~  
File Edit View Terminal Tabs Help  
user@user-VirtualBox:~$ % ps -e -o pid, ppid, command  
bash: fg: %: no such job  
user@user-VirtualBox:~$ ps -e -o pid, ppid, command  
PID    PPID  COMMAND  
1       0    /sbin/init splash  
2       0    [kthreadd]  
3       2    [rcu_gp]  
4       2    [rcu_par_gp]  
6       2    [kworker/0:0H-events_highpri]  
7       2    [kworker/0:1-events]  
9       2    [mm_percpu_wq]  
10      2    [rcu_tasks_rude_]  
11      2    [rcu_tasks_trace]  
12      2    [ksoftirqd/0]  
13      2    [rcu_sched]  
14      2    [migration/0]  
15      2    [idle_inject/0]  
16      2    [cpuhp/0]  
17      2    [cpuhp/1]  
18      2    [idle_inject/1]  
19      2    [migration/1]  
20      2    [ksoftirqd/1]  
22      2    [kworker/1:0H-events_highpri]  
23      2    [cpuhp/2]  
24      2    [idle_inject/2]  
25      2    [migration/2]  
26      2    [ksoftirqd/2]  
27      2    [kworker/2:0-events]  
28      2    [kworker/2:0H-events_highpri]  
29      2    [cpuhp/3]  
30      2    [idle_inject/3]  
31      2    [migration/3]  
32      2    [ksoftirqd/3]  
34      2    [kworker/3:0H-events_highpri]  
35      2    [kdevtmpfs]  
36      2    [netns]  
37      2    [inet_frag_wq]  
38      2    [kauditd]  
39      2    [khungtaskd]  
40      2    [oom_reaper]  
41      2    [writeback]  
42      2    [kcompactd0]  
43      2    [ksmd]  
44      2    [khugepaged]  
51      2    [kworker/1:1-events]  
91      2    [kintegrityd]  
92      2    [kblockd]  
93      2    [blkcg_punt_bio]  
94      2    [tpm_dev_wq]  
95      2    [ata_sff]  
96      2    [md]  
97      2    [edac-poller]  
98      2    [devfreq_wq]  
99      2    [watchdogd]  
100     2    [kworker/3:1-events]  
102     2    [kworker/0:1H-kblockd]  
104     2    [kworker/2:1-mm_percpu_wq]  
105     2    [kswapd0]  
106     2    [ecryptfs-kthrea]  
108     2    [kthrotld]  
109     2    [acpi_thermal_pm]  
110     2    [scsi_eh_0]  
111     2    [scsi_tmf_0]  
112     2    [scsi_eh_1]  
113     2    [scsi_tmf_1]  
114     2    [kworker/u8:2-ext4-rsv-conversion]
```

3.1.3 Matando um processo

Você pode matar um processo em execução com o comando kill. Basta especificar na linha de comando o ID do processo a ser eliminado. O comando kill funciona enviando ao processo um SIGTERM, ou terminação, signal.1 Isso faz com que o processo termine, a menos que o programa em execução explicitamente manipula ou mascara o sinal SIGTERM.

3.2 Criando processos

Duas técnicas comuns são usadas para criar um novo processo. Simples, mas deve ser usado com moderação porque é ineficiente e tem consideravelmente riscos de segurança. A segunda técnica é mais complexa, mas oferece maior flexibilidade, velocidade e segurança

3.2.1 Usando System

A função do sistema na biblioteca C padrão fornece uma maneira fácil de executar um comando de dentro de um programa, como se o comando tivesse sido digitado em um Concha. Na verdade, o sistema cria um subprocesso executando o shell Bourne padrão (/bin/sh) e entrega o comando a esse shell para execução.

A função do sistema retorna o status de saída do comando shell. Se a própria casca não pode ser executado, o sistema retorna 127, se ocorrer outro erro, o sistema retornará -1. Como a função do sistema usa um shell para invocar seu comando, ela está sujeita a os recursos, limitações e falhas de segurança do shell do sistema. disponibilidade de qualquer versão específica do shell Bourne.

3.2.2 Usando fork and exec

O Linux fornece uma função, fork, que cria um processo filho que é um 3.2 Criando Processos 49 cópia de seu processo pai. O Linux fornece outro conjunto de funções, a família exec, que faz com que um determinado processo deixe de ser uma instância de um programa e, em vez disso, se tornar uma instância de outro programa. Para gerar um novo processo, você primeiro usa fork para fazer uma cópia do processo atual.

Chamando fork

Quando um programa chama fork, um processo duplicado, chamado de processo filho, é criado processo pai continua executando o programa a partir do ponto em que fork foi chamado. O processo filho também executa o mesmo programa no mesmo lugar. Então, como os dois processos diferem? Primeiro, o processo filho é um processo novo e portanto, tem um novo ID de processo, distinto do ID de processo de seu pai.

3.2.3. Processo Scheduling

O Linux agenda os processos pai e filho independentemente; não há garantia de qual será executado primeiro, ou por quanto tempo ele será executado antes que o Linux o interrompa e deixe o outro processo (ou algum outro processo no sistema) ser executado. Em particular, o comando ls pode ser executado no processo filho antes que o pai seja concluído. Mas, pode-se especificar que um processo é menos importante - e deve

receber uma prioridade mais baixa - atribuindo-lhe um valor de gentileza mais alto. Por padrão, todo processo tem uma gentileza de zero.

3.3. Sinais

Um sinal é uma mensagem especial enviada a um processo. Os sinais são assíncronos, quando um processo recebe um sinal, ele processa o sinal imediatamente. Cada tipo de sinal é especificado por seu número de sinal, mas em programas, geralmente se refere a um sinal pelo nome. No Linux, eles são definidos em `/usr/include/bits/signum.h`.

Quando um processo recebe um sinal, ele pode fazer uma de várias coisas, dependendo da disposição do sinal. Para cada sinal, há uma disposição padrão, que determina qual acontece com o processo se o programa não especificar algum outro comportamento. Para a maioria dos tipos de sinal, um programa pode especificar algum outro comportamento - tanto para ignorar o sinal ou para chamar uma função especial de tratamento de sinal para responder ao sinal. Se um manipulador de sinal for usado, o programa atualmente em execução é pausado, o manipulador de sinais é executado e, quando o manipulador de sinal retorna, o programa é retomado.

3.4. Processo Termination

Normalmente, um processo termina de duas maneiras. Ou o programa em execução chama a função `exit`, ou a função `main` do programa retorna. Cada processo tem uma saída: um número que o processo retorna ao seu pai. O código de saída é o argumento passado para a função `exit` ou o valor retornado de `main`.

3.4.1. Esperando pelo processo termination

A saída do programa `ls` geralmente aparece após o “programa principal” que já foi concluído. Isso porque o processo filho, no qual `ls` é executado, é escalonado independentemente do processo pai. Como o Linux é um sistema operacional multitarefas, ambos os processos parecem ser executados simultaneamente e você não pode prever se o programa `ls` terá a chance de ser executado antes ou depois da execução do processo pai.

3.4.2. A espera das chamadas de sistema

A função mais simples é chamada simplesmente `wait`. Ela bloqueia o processo de chamada até que um de seus processos filho é encerrado (ou ocorre um erro). Ele retorna um código de status por meio de um argumento inteiro de ponteiro, do qual você pode extrair informações sobre como o processo filho saiu.

3.4.3. Processos zumbis

Um processo zumbi é um processo que terminou, mas ainda não foi limpo. Assim, é responsabilidade do processo pai terminar seu filho (processo zumbi). A espera de funções também faz isso, então não é necessário rastrear se seu processo filho ainda está executando antes de esperar por ele. Suponha, por exemplo, que um programa bifurque um processo filho, executando alguns outros cálculos e, em seguida, chama wait. Se o processo filho não tiver terminado nesse ponto, o processo pai bloqueará a chamada de espera até que o processo filho termine. Se o processo filho terminar antes que o processo pai chame wait, o processo filho se torna um zumbi. Quando o processo pai chama wait, o status zumbi de encerramento do filho é extraído, o processo filho é excluído e a chamada de espera retorna imediatamente.

3.4.4. Limpeza processos filhos assíncronos

Uma maneira fácil de limpar processos filho é manipular SIGCHLD. Claro, ao limpar o processo filho, é importante armazenar seu status de encerramento se essa informação for necessária, porque uma vez que o processo é limpo usando wait, essa informação não estará mais disponível.

Resumo tópicos vídeo – Linux Kernel Development

1. About Linus and Linux Creation

A primeira versão do Linux foi desenvolvida em sua primeira versão no ano de 1991 por Linus Torvald, que começou a escrever um simples terminal emulado. Atualmente, o Linux é um sistema open source, ou seja, seus códigos internos são liberados para todos que quiserem se aprofundar e entender como tudo funciona, e é bem adaptado tanto para pequenas quanto para grandes máquinas. O sistema Linux é baseado em um sistema Unix, mas não é o próprio Unix.

2. How Linux Kernel was same and different to other kernels when it was created

Os kernels, podem basicamente serem divididos em duas escolas diferentes, o kernel monolítico e o microkernel, onde temos o monolítico como o mais simples, e todos os kernels foram projetados dessa maneira até a década de 1980. Ele é implementado em um único processo e em um único espaço de endereço e existem no disco como binários estáticos únicos, então todos os serviços do kernel existem e executam no grande espaço de endereço do kernel e a comunicação dentro do kernel é trivial porque tudo funciona em modo kernel no mesmo espaço de endereço. Os microkernel, por outro lado, não são implementados como um único grande processo, em vez da funcionalidade do kernel é dividido em processos separados, e são chamados de servidores, separados em diferentes espaços de endereços, assim a invocação direta não é possível como no monolítico. O Linux possui um kernel monolítico e suporta carregamentos dinâmicos de módulos de kernel, além de ser preemptivo. O Linux também, é grátis em todos os sentidos no mundo.

5. How coding inside the kernel is different then coding in user space

O espaço do usuário é um sistema de memória que está alocado para funcionar aplicativos para que a memória virtual seja dividida em espaço do kernel e o espaço do usuário. O espaço do kernel é aquela área virtual onde os processos do kernel serão executados e o espaço do usuário é aquela área virtual de memória onde os processos do usuário serão executados. Esta divisão é necessária para a memória e proteção de acesso.

O kernel não tem acesso nem a biblioteca nem ao cabeçalho padrão enquanto o kernel é codificado em ge uc tao parecido com qualquer kernel unix que se preze. O kernel do Linux é programado em c. O kernel não é programado em estrito ncc em vez de onde aplicável. O kernel carece de proteção de memória oferecido ao espaço do usuário quando um aplicativo de espaço do usuário tenta acesso ilegal à memória o kernel pode capturar o erro e enviar o sinal e matar o processo. Como o Linux é preemptivo, sem proteção, o código do kernel pode ser preventivo em favor de um código diferente do acesso ao mesmo recurso.

6. How processes are tracked and managed in kernel

Um processo é basicamente um programa que é no processo de estado de execução são mais do que apenas o código do programa em execução, eles também incluem um conjunto de recursos como arquivos abertos e sinaliza qualquer programa que está em estado de execução, denominado processo.

O estado do processo é basicamente o status atual do processo e existem vários estados. Um deles é a tarefa em execução, e esta é uma situação quando adquiriu algum recurso do sistema e está em execução ou está em fase e está gerando alguma saída e tomando qualquer entrada. O processo interrompível é quando o processo está dormindo ou esperando que alguma condição exista. Temos também o estado ininterrupto, que é igual ao interrompível, mas nessa condição o processo não é acordado e não se torna executável se receber um sinal.

7. Threads in Linux

Um thread é basicamente a menor unidade de processamento que pode ser realizado em um sistema operacional, no mais moderno sistema operacional existe um thread dentro de um processo que um único processo pode conter vários tópicos. Então quando temos que fazer varias tarefas em um único processo, então criamos um thread para cada processo e os threads são executados simultaneamente alcançando nosso processo. O Linux não considera o thread por cada thread, ou seja, cada thread é considerado um processo separado no kernel. O kernel do Linux não fornece nenhuma semântica ou dados de agendamento especial e estrutura para representar os threads, em vez disso, um thread é um processo que realiza certas ações e recursos com outros processos.

8. Process Scheduling and Scheduling Algorithms

A politica do agendador geralmente determina a sensação geral de um sistema e é responsável por otimizar utilizando tempo do processador, portanto, é muito importante ter uma boa política quando vocês estão fazendo qualquer sistema operacional. Então a primeira politica que temos no kernel é o i vinculado ao limite do processador, Processos podem ser classificados como io limite e processador vinculado. O processo que gasta muito esta na hora de enviar e esperar o i o pedido e é chamado o processo i e ligado ao processo vinculado ao processador, e isso gasta muito tempo na execução. Então quando tivermos qualquer processo que é vinculado como se precisasse de algum recurso de entrada e saída, então nos o chamamos de processo ligado a io e o outro processo que tem algumas execuções do código e esta executando o código no kernel por muito tempo, então nos o chamamos de processo vinculado ao processador. Esse processo é executado até eles serem antecipados porque não bloqueiam no io e o request muitas vezes não são io 1 no entanto a resposta do sistema não determina o agendador executa-los depois.

A politica de agendamento em sistema deve tentar satisfazer dois objetivos conflitantes, tempo de resposta do processo rápido que chamamos de baixa latência, então

quando o processo for rápido, ele terá baixa latência e o outro é o sistema máximo de utilização. Para satisfazer esses requisitos, planejadores frequentemente empregam algoritmos complexos para determinar os processos mais valiosos para correr sem comprometer a justiça para com os outros processos de menor prioridade.

9. What is a System Call, how to call them

Este tópico aborda as chamadas de sistema e como são essas chamadas nos sistemas operacionais modernos. O Kernel fornece um conjunto de interfaces para saber quais processos estão em execução na camada do usuário. O espaço do usuário pode interagir com o sistema operacional e é através das chamadas de sistema que a interação é possível de acontecer, sendo que essas interfaces atuam como um "mensageiro" entre as aplicações da camada de usuário e o kernel. Assim, a chamada de sistema fornece uma camada entre o hardware e a interface do usuário solicitando um serviço do kernel.

10. System Call implementation in the kernel

O presente tópico aborda sobre a implementação de chamada de sistema no kernel, sendo que a implementação real no linux não precisa ser relacionado com o comportamento do gerenciador de chamadas de sistema, pois adicionar uma nova chamada de sistema ao linux é relativamente fácil. A parte difícil está em projetar e implementar a chamada de sistema. O primeiro passo deste processo é definir um sistema de multiplexação, onde a chamada de sistema deve possuir uma interface simples com o menor número de variáveis possíveis. Outro passo a ser considerado são os parâmetros das chamadas de sistema, que devem ser verificados cuidadosamente para garantir que eles sejam válidos e que não estejam compartilhando alguns dados que podem ser considerados como um comportamento ilegal.

12. what is an interrupt and how they are handled in kernel

13. What is an IRQ?

Agora, estes tópicos que estão relacionados entre si, abordam sobre o que é uma interrupção e como elas são tratadas no kernel. Uma interrupção é uma resposta do processador a um evento que precisa de atenção do software, por exemplo, pressionar alguma tecla do teclado e o processador ser notificado sobre essa ação. Assim, ao clicarmos na tecla o processador estará gerando um evento e perguntará ao software para assim dar uma resposta sobre tal evento. Então, esse ato de pressionar a tecla é basicamente uma interrupção de responsabilidade central de qualquer kernel do sistema operacional. O kernel precisa se comunicar com dispositivos individuais da máquina como se tivéssemos vários dispositivos de hardware em nosso sistema operacional. Assim, o kernel tem que gerenciar todos esses dispositivos de hardware e sempre que algum trabalho for atribuído ao processador ou hardware pelo kernel, esses dispositivos irão sinalizar para o kernel a situação de conformidade sobre realizar tal tarefa e este sinal é chamado de interrupção de ativação.

15. About critical regions and race conditions, how to protect?

O presente tópico aborda sobre as regiões críticas, condições de corrida e como protegê-las. Em todos os sistemas operacionais existe essa região crítica e a ocorrência da condição de corrida, sendo que a região crítica é basicamente um caminho de código que acessa e consegue manipular dados restritos. Assim, nessa região apenas um processo por vez tem permissão para realizar a tarefa, mas se houver mais de um processo que está tentando acessar a região crítica ocorre o evento que é denominado de condição de corrida.

17. Understanding Kernel Notion of Time

A noção central de tempo é como o kernel gerencia o tempo e como ele mantém o tempo atualizado. Um grande número de funções do kernel são acionadas pelo tempo. Sempre que o tempo passa uma função é acionada. Algumas dessas funções são periódicas como balancear o escalonador ou atualizando a tela, eles ocorrem em uma programação fixa. O kernel agenda outras funções como o disco de atraso io no tempo relativo no futuro. Um kernel pode agendar o trabalho para 500 milissegundos a partir de agora, finalmente, o kernel também deve gerenciar o sistema de tempo e a data e hora atuais.

Acompanhar o tempo de atividade do Linux também é a função que é executada pelo kernel do Linux e o kernel também gerencia o atual data e hora no seu sistema, por isso é muito importante para o kernel acompanhar o tempo. A diferença entre o tempo relativo e absoluto é notável.

19. Kernel Memory Management Theory

O kernel não pode facilmente lidar com alocação de memória, muitas vezes ele não consegue dormir por causa dessas limitações e a necessidade de uma memória leve e esquema de alocação. Obter memória no kernel é mais complicado do que no espaço do usuário. Do ponto de vista de um programador de kernel, as alocações de memória são difíceis. O kernel trata páginas físicas como unidades básicas de gerenciamento de memória em sistemas operacionais, e a paginação é um esquema de gerenciamento de memória pelo qual um computador armazena e recupera dados de armazenamento secundário para uso na memória principal.

O gerenciamento de memória normalmente trata de páginas, portanto, a unidade de gerenciamento tenta manter a tabela de paginas do sistema com páginas do tamanho necessário para que o sistema ou kernel possa gerenciar, em seguida isso faz basicamente uma tabela da pagina de tamanho. Muitas arquiteturas até suportam múltiplos tamanhos de página, mas a maioria das arquiteturas de 32 bits tem quatro kilobytes de pagina enquanto a maioria das arquiteturas de 64 bits tem paginas de oito kilobytes.

24. Filesystem Abstraction Layer

Por fim, temos o tópico que aborda sobre a camada de abstração do sistema de arquivos. O kernel do linux implementa uma camada de abstração em torno de seu baixo nível, onde a interface do sistema poderia ter mais de um sistema de arquivos. No linux, existe uma interface genérica para qualquer tipo de sistema de arquivos que apenas é viável porque o kernel implementa uma camada de abstração em torno de seu baixo nível. Uma camada de abstração ou nível de abstração é uma maneira de esconder os detalhes do trabalho de um subsistema e permite a separação de interesses. Então, se o kernel implementar um sistema de arquivo múltiplo será difícil acessar de acessas esses sistemas, pois teremos que implementar uma inferface separada para tal aplicação.