



## **Sistemas Embarcados 2**

### **Semana 03**

Professor: Éder Moura

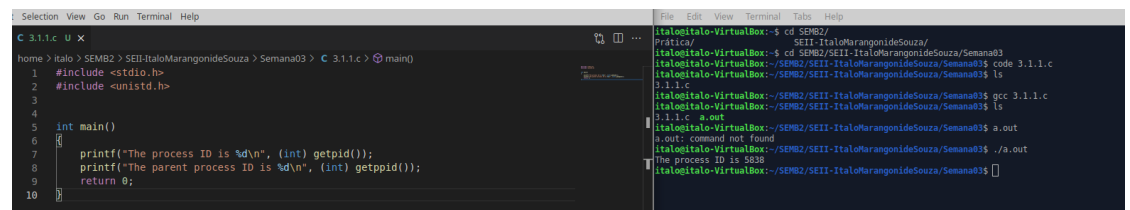
Aluno: Ítalo Marangoni De Souza

11811EAU014

### 3. FAÇA UM RESUMO DE TODAS AS SEÇÕES DO CAPÍTULO 3, DO LIVRO ADVANCED LINUX PROGRAMMING, E IMPLEMENTE OS EXEMPLOS DISPONIBILIZADOS.

- 3.1.1 Um único processo identifica cada processo no sistema Linux, pode ser referido também como *pid*. Sendo identificados em números de 16 bits.

Todo processo tem um processo parente (exceto o init ou “zombie process”). Dessa forma, podemos fazer a analogia como se fosse uma árvore, onde sempre haverá uma ramificação de um processo raiz.



```
Selection View Go Run Terminal Help
C 3.1.1.c U X
home > italo > SEMB2 > SEII-ItaloMarangonideSouza > Semana03 > C 3.1.1.c > main()
1 #include <stdio.h>
2 #include <unistd.h>
3
4
5 int main()
6 {
7     printf("The process ID is %d\n", (int) getpid());
8     printf("The parent process ID is %d\n", (int) getppid());
9     return 0;
10 }

Italo@italo-VirtualBox:~$ cd SEMB2/
Italo@italo-VirtualBox:~$ cd SEMB2/SEII-ItaloMarangonideSouza/Semana03
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$ code 3.1.1.c
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$ ls
3.1.1.c
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.1.1.c
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$ ls
3.1.1.c a.out
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$ a.out
a.out: command not found
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$ ./a.out
The process ID is 9508
Italo@italo-VirtualBox:~/SEII-ItaloMarangonideSouza/Semana03$
```

- 3.1.2 O comando *ps* mostra os comandos que estão “rodando” no seu sistema. Os processos que serão exibidos no painel, primeiro será o bash, depois o *ps program* e outros processos subsequentes.
- 3.1.3 O comando *kill* serve para você “matar” um processo. Apenas digitar o comando junto com o ID do processo em questão, assim, o processo recebe o comando para terminar a tarefa.
- 3.2 Para criar novos processos, há duas técnicas comuns. A primeira, é simples relativamente, porém possui riscos de segurança e deve ser usado de modo moderado. A segunda é um pouco mais complexa, garante mais flexibilidade, velocidade e segurança.
- 3.2.1 Usando uma função do sistema em biblioteca C, esse caminho garante que a execução de um comando proveniente de um programa, podendo ser digitado dentro do *shell*.

```

C 3.1.1.c U C 3.2.1.c U X
home > italo > SEMB2 > SEII-ItaloMarangonideSouza > Semana03 > C 3.2.1.c > main()
1 #include <stdlib.h>
2
3
4 int main()
5 {
6     int return_value;
7     return_value = system("ls -l /");
8     return return_value;
9 }

Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ cd SEMB2/SEII-ItaloMarangonideSouza/Semana03
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.1.1.c
3.1.1.c: a.out
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.1.1.c
3.1.1.c: a.out
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ ./a.out
a.out: command not found
The process ID is 5838
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.2.1
gcc: error: 3.2.1: No such file or directory
gcc: fatal error: no input files
compilation terminated.
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.2.1.c
3.1.1.c: 3.2.1.c: a.out
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ ./a.out
total 76
lnxhexnxx 1 root root 7 dez 6 14:51 bin -> usr/bin
dlnxhexnxx 3 root root 4096 jan 5 18:50 boot
dlnxhexnxx 2 root root 4096 dez 6 14:53 cdrom
dlnxhexnxx 20 root root 4120 jan 10 16:57 dev
dlnxhexnxx 134 root root 12288 jan 5 08:09 etc
dlnxhexnxx 3 root root 4096 dez 6 14:54 home
lnxhexnxx 1 root root 7 dez 6 14:51 lib -> usr/lib
lnxhexnxx 1 root root 9 dez 6 14:51 lib32 -> usr/lib32
lnxhexnxx 1 root root 9 dez 6 14:51 lib64 -> usr/lib64
lnxhexnxx 1 root root 10 dez 6 14:51 libx32 -> usr/libx32
dlnxhexnxx 2 root root 16384 dez 6 14:50 lost+found
dlnxhexnxx 3 root root 4096 dez 6 15:06 media
dlnxhexnxx 2 root root 4096 ago 19 07:29 mnt
dlnxhexnxx 3 root root 4096 dez 6 15:17 opt
dlnxhexnxx 267 root root 0 jan 10 16:57 proc
dlnxhexnxx 1 root root 4096 dez 6 23:01 root
dlnxhexnxx 27 root root 840 jan 10 17:45 run
lnxhexnxx 1 root root 8 dez 6 14:51/sbin -> usr/sbin
dlnxhexnxx 2 root root 4096 dez 6 15:07 snap
dlnxhexnxx 2 root root 4096 ago 19 07:29 srv
dlnxhexnxx 13 root root 0 jan 10 16:57 sys
dlnxhexnxx 10 root root 4096 jan 10 17:56 tmp
dlnxhexnxx 14 root root 4096 ago 19 07:32 usr
dlnxhexnxx 14 root root 4096 ago 19 07:43 var
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$

```

O sistema retorna o status de saída do comando shell, caso o shell não consiga executar, o sistema retorna 127, caso outro erro ocorra, o sistema retorna -1.

3.2.2 A função fork é fornecida pelo Linux, ela cria um processo filho exatamente igual ao pai.

Ao chamar fork, um processo é duplicado, chamado assim de processo filho.

```

C 3.1.1.c U C 3.2.1.c U C 3.2.2.c X
home > italo > SEMB2 > SEII-ItaloMarangonideSouza > Semana03 > C 3.2.2.c > ...
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     pid_t child_pid;
8
9     printf("the main program process ID is %d\n", (int) getpid());
10    child_pid = fork();
11
12    if (child_pid != 0) {
13        printf("this is the parent process with ID %d\n", (int) getpid());
14        printf("the child's process ID is %d\n", (int) child_pid);
15    }
16    else {
17        printf("this is the child process with ID %d\n", (int) getpid());
18    }
19    return 0;
20 }
21 }

Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.2.2.c
3.1.1.c 3.2.1.c 3.2.2.c
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.2.2.c
3.1.1.c 3.2.1.c 3.2.2.c
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ ./a.out
collect2: error: ld returned 1 exit status
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$ gcc 3.2.2.c
The main program process ID is 6035
this is the parent process with ID 6035
this is the child process with ID 6036
Italo@italo-VirtualBox:~/SEMB2/SEII-ItaloMarangonideSouza/Semana03$

```

A função exec tem como objetivo substituir o programa que está rodando em um processo por um outro programa.

3.2.3 Os programas parentes e filhos são programados independentemente pelo Linux. Não havendo garantia de qual ocorrerá primeiro, ou por quanto tempo. Mesmo assim, pode definir o grau de importância, fornecendo grau de menor prioridade.

3.3 *Signals* são mecanismos que servem para comunicação com manipulação de processos Linux. Um signal é uma mensagem especial enviada a um processo, estes são assíncronos. Ao receber um signal, um processo o processa imediatamente, sem precisar terminar a tarefa anterior.

```

C 3.1.c U C 3.2.1.c U C 3.2.2.c U C 3.3.c U X
home > italo > SEMB2 > SEII-ItaloMarangonideSouza > Semana03 > C 3.3.c > (main)
1 #include<signal.h>
2 #include<stdio.h>
3 #include<string.h>
4 #include<sys/types.h>
5 #include<unistd.h>
6
7 sig_atomic_t sigusr1_count = 0;
8
9 void handler(int signal_number)
10 {
11     ++sigusr1_count;
12 }
13
14 int main()
15 {
16     struct sigaction sa;
17     memset(&sa, 0, sizeof(sa));
18     sa.sa_handler = &handler;
19     sigaction(SIGUSR1, &sa, NULL);
20
21     printf("SIGUSR1 was raised %d times\n", sigusr1_count);
22     return 0;
23 }

```

```

Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ ls
3.1.1.c 3.2.1.c 3.2.2.c
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ gcc 3.2.2.c
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/libc.so.2: undefined reference to 'main'
collect2: error: ld returned 1 exit status
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ gcc 3.2.2.c
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ ./a.out
the main program process ID is 6035
this is the child process with ID 6035
this is the child process with ID 6036
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ gcc 3.3.c
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ ./a.out
SIGUSR1 was raised 0 times
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$

```

- 3.4 Um processo termina, normalmente, em uma de duas maneiras. Ambas executam um chamado para fim da função. Um processo pode ser terminado via signal, por exemplo, SIGBUS, SIGSEV e SEGVPE são sinais que finalizam um processo.

```

C 3.1.c U C 3.2.1.c U C 3.2.2.c U C 3.3.c U C 3.4.c X
home > italo > SEMB2 > SEII-ItaloMarangonideSouza > Semana03 > C 3.4.c > (main)
1 #include<signal.h>
2 #include<stdio.h>
3 #include<string.h>
4 #include<sys/types.h>
5 #include<unistd.h>
6 #include<stdlib.h>
7
8
9 int main()
10 {
11     int child_status;
12     char* arg_list[]={
13         "ls",
14         "-l",
15         "/",
16         NULL
17     };
18
19     spawn("ls", arg_list);
20
21     wait(&child_status);
22     if(WIFEXITED(child_status))
23         printf("the child process exited normally, with exit code %d\n", WEXITSTATUS(c
24     else
25         printf("the child process exited abnormally\n");
26
27     return 0;
28 }

```

```

Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$ gcc 3.4.c
3.4.c:19: warning: implicit declaration of function 'spawn' [-Wimplicit-function-declaration]
20 | spawn("ls", arg_list);
   | ^~~~~
3.4.c:22:2: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
22 | wait(&child_status);
   | ^~~~
/usr/bin/ld: /tmp/cc0w9W.o: in function 'main':
3.4.c:(.text+0x5): undefined reference to 'spawn'
collect2: error: ld returned 1 exit status
Italo@italo-VirtualBox: ~/SEMI-ItaloMarangonideSouza/Semana03$

```

Um exemplo do comando ls, primeiro é executado corretamente e retorna 0, segundo é apresentado um erro e retorna 1.

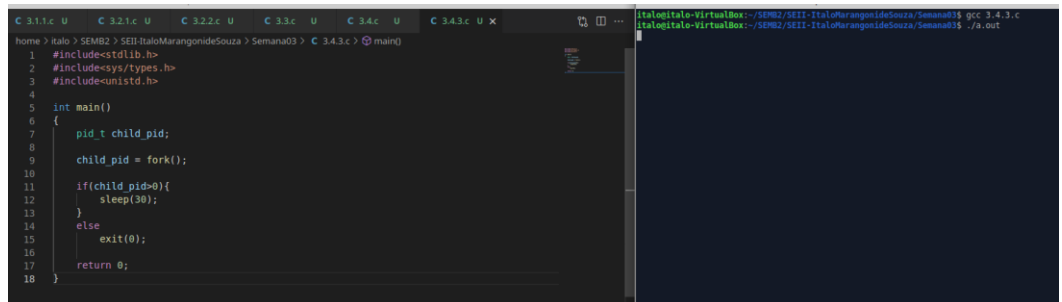
- 3.4.1 Ao executar os comandos fork e exec, você não pode ter notado que a saída do programa ls, depois do “main program” já foi concluída. Isso ocorre porque o processo filho é agendado independente do processo pai.
- 3.4.2 A função wait, nos retorna um código de status por meio de um argumento em ponteiro inteiro, no qual podem se extrair informações sobre o filho encerrado.

Usando a macro WIFEXITED podemos determinar a partir do status de saída de um processo filho se esse processo saiu normalmente.

- 3.4.3 Caso um processo filho termine enquanto um processo pai está chamando uma função de espera, o filho desaparece e seu status de término é passado para o processo pai por uma wait call.

No caso de um processo filho terminar e o pai não estiver chamado um wait, o filho passa a ser denominado zombie.

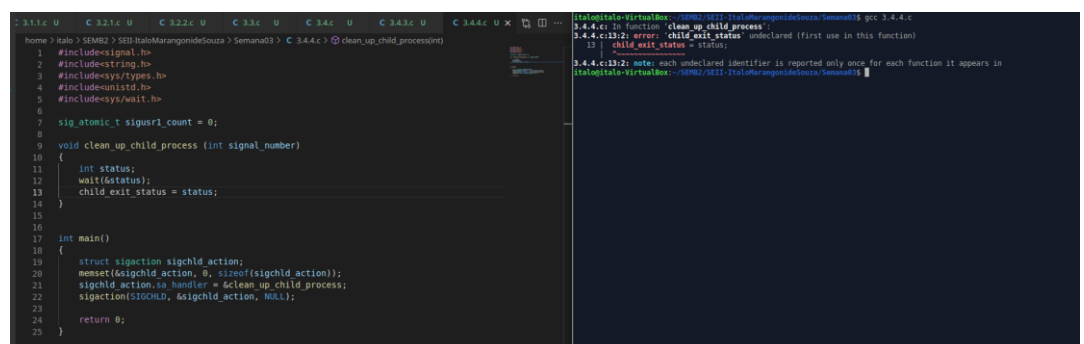
Zombie é um processo que foi encerrado, mas ainda não foi limpo. O processo pai fica responsável por limpar o zombie child.



```
home > italo > SEMB2 > SEII-ItaloMarangoniSouza > Semana03 > C 3.4.3.c > main()
1 #include<stdlib.h>
2 #include<sys/types.h>
3 #include<unistd.h>
4
5 int main()
6 {
7     pid_t child_pid;
8
9     child_pid = fork();
10
11     if(child_pid>0){
12         sleep(30);
13     }
14     else
15         exit(0);
16
17     return 0;
18 }
```

3.4.4 Se estiver usando um processo filho simplesmente para executar outro programa, não há problema chamar wait imediatamente no processo pai, que será bloqueado até que o processo filho seja concluído.

Dessa forma, uma maneira fácil de limpar os processos filhos é manipulando SIGCHLD. Ao limpar o processo filho, é importante armazenar os seus status de encerramento, caso a informação for necessária, pois uma vez que é limpo usando wait, a informação não fica mais disponível.



```
home > italo > SEMB2 > SEII-ItaloMarangoniSouza > Semana03 > C 3.4.4.c > clean_up_child_process()
1 #include<signal.h>
2 #include<string.h>
3 #include<sys/types.h>
4 #include<unistd.h>
5 #include<sys/wait.h>
6
7 sig_atomic_t sigusr1_count = 0;
8
9 void clean_up_child_process (int signal_number)
10 {
11     int status;
12     wait(&status);
13     child_exit_status = status;
14 }
15
16
17 int main()
18 {
19     struct sigaction sigchld_action;
20     memset(&sigchld_action, 0, sizeof(sigchld_action));
21     sigchld_action.sa_handler = &clean_up_child_process;
22     sigaction(SIGCHLD, &sigchld_action, NULL);
23
24     return 0;
25 }
```

## 4. FAÇA UM RESUMO DOS TÓPICOS APRESENTADOS NAS SEQUENTES SEÇÕES... (VÍDEO NO YOUTUBE).

### 1. ABOUT LINUS AND LINUX CREATION

Linus Torvald foi um estudante da Universidade de Helsinki e desenvolveu a primeira versão do Linux em 1991. Inicialmente o sistema era um emulador de terminal, ele escreveu um programa especificamente para o hardware que estava usando e independente de um sistema operacional, porque queria usar as funções de seu novo computador com um processador 80386.

## 2. HOW LINUX KERNEL WAS SAME AND DIFFERENT TO OTHER KERNELS WHEN IT WAS CREATED

O kernel do Linux é de código aberto baseado no Unix, portanto, todos tem acesso ao código fonte. Isso significa que qualquer pessoa pode trabalhar no desenvolvimento, e é livre para usar como quiser.

O kernel do Linux possui arquitetura monolítica, além de manter suas configurações na forma de arquivos e permite um ambiente multiusuário.

## 5. HOW CODING INSIDE THE KERNEL IS DIFFERENT THEN CODING IN USER SPACE

O espaço do kernel é onde o núcleo do sistema operacional funciona e fornece seus serviços. É algo que o usuário não pode interferir. Já o espaço do usuário é a parte da memória do sistema em que os processos são gerenciados pelo kernel.

As principais diferenças são: no kernel não há bibliotecas ou cabeçalhos padrão; no kernel o código é feito somente em linguagem C; no kernel o espaço de memória é limitado, portanto alguns processos rodados nesse local podem apresentar “kill the process” devido à falta de memória.

## 6. HOW PROCESSES ARE TRACKED AND MANAGED IN KERNEL

O processo atual deve ser processado antes que outro processo possa ser selecionado para execução. Se a política de agendamento dos processos atuais for “redonda”, então ela será colocada na parte de trás da fila de execução. Se a tarefa for interrompida e receber um *signal* a última vez que foi agendada, seu estado se torna RUNNING.

Se o processo atual tiver sido cronometrado, então seu estado se tornará RUNNING. Se o processo atual estiver em execução, então ele permanecerá nesse estado.

Processos que não estavam funcionando nem interrupções são removidos da fila de execução. Isso significa que eles não serão considerados para correr quando o agendador procurar o processo mais merecido para executar.

## 7. THREADS IN LINUX

O Linux tem uma única implementação de threads. O kernel do Linux não fornece nenhuma semântica de agendamento especial ou estruturas de dados para representar threads. Em vez disso, a ideia de thread é apenas um processo que compartilha certos recursos com outros processos.

Cada segmento tem uma `task_struct` única e aparece no kernel como um processo normal.

## 8. PROCESS SCHEDULING AND SCHEDULING ALGORITHMS

As políticas de agendamento são regras que o agendador segue para determinar o que deve ser executado e quando. Essa política considera dois processos: os vinculados à I/O e os processos vinculados à CPU.

Processos vinculados a I/O passam a maior parte do tempo esperando que as operações I/O, como solicitação de rede ou operação de teclado, sejam concluídas.

Já processos vinculados à CPU passam a maior parte do tempo executando o código. Estes, são frequentemente antecipados porque bloqueiam muitas vezes as solicitações de I/O.

O kernel usa dois valores prioritários separados. Um bom valor (-20 a +19), e um valor prioritário em tempo real (0 a 99). Para o primeiro caso, quanto maior o valor menos a prioridades, já no segundo caso a ideia é contrária, quanto maior o valor maior é a prioridade.

O *timeslice* representa quanto tempo um processo pode ser executado antes de ser antecipado. A política do agendador deve decidir sobre um timeslice padrão.

## 9. WHAT IS A SYSTEM CALL, HOW TO CALL THEM

Chamadas do sistema são como um programa entra no kernel para executar alguma tarefa. Os programas usam chamadas do sistema para executar uma variedade de operações como: criação de processos, fazer IO de rede e arquivo, entre outros. Essas chamadas de sistema variam entre as arquiteturas da CPU.

## 10. SYSTEM CALL IMPLEMENTATION IN THE KERNEL

A chamada de sistema é implementada por uma “interrupção de software” que transfere o controle para o código do kernel. A chamada específica do sistema que está sendo invocada é armazenada no registro, seus argumentos são mantidos nos outros registros de processadores.

Após a mudança para o kernel, o processador deve salvar todos os seus registros e despachar a execução para função adequada do kernel, depois verificar se está fora de alcance.

## 12. WHAT IS AN INTERRUPT AND HOW THEY ARE HANDLED IN KERNEL

Uma interrupção é um evento que altera o fluxo normal de execução de um programa e pode ser gerado por dispositivos de hardware ou até mesmo pela própria CPU. Quando uma interrupção ocorre o fluxo atual de execução é suspenso e interrompe as corridas do manipulador. Depois que o manipulador de interrupção executa o fluxo de execução anterior é retomado.

## 13. WHAT IS AN IRQ?

Um dispositivo que suporta interrupções tem um pino de saída usado para sinalizar um Interrupt ReQuest (IRQ). Os pinos IRQ estão conectados a um dispositivo chamado Programmable Interrupt Controller (PIC).



## 15. ABOUT CRITICAL REGIONS AND RACE CONDITIONS, HOW TO PROTECT?

Um *race condition* geralmente envolvem um ou mais processos acessando um recurso compartilhado (arquivo ou variável), onde esse acesso múltiplo não foi devidamente controlado. Os *race conditions* podem ser definidos como:

*Interferência causa por processos não confiáveis.* Algumas taxonomias de segurança chamam esse problema de condição de “sequência” ou “não atômica”. Essas são condições causadas por processos em execução de outros programas diferentes.

*Interferência causa por processos confiáveis.* Algumas taxonomias chamam esses impasses, *livelock* ou condições de falha de bloqueio. Estas condições são causadas por processos que executam o mesmo programa.

## 17. UNDERSTANDING KERNEL NOTION OF TIME

O tempo de processo é definido como a quantidade de tempo de CPU usado por um processo. Isso às vezes é dividido em usuário e componentes de sistema. O tempo de CPU do sistema é o tempo gasto pelo kernel executando em modo de sistema em nome do processo.

A maioria dos computadores tem um relógio de hardware (alimentado por bateria) que o kernel lê no momento de inicialização para inicializar o software.

## 19. KERNEL MEMORY MANAGEMENT THEORY

O sistema operacional faz com que o sistema pareça como se tivesse uma quantidade maior de memória do que realmente tem. A memória virtual pode ser muitas vezes maior do que a memória física no sistema. Cada processo no sistema tem seu próprio espaço de endereço virtual. O mapeamento de memória é usado para mapear arquivos de imagem e dados em um espaço de endereço de processos. No mapeamento de memória, o conteúdo de um arquivo está ligado diretamente ao espaço de endereço virtual de um processo.

## 24. FILESYSTEM ABSTRACTION LAYER

Tal interface genérica para qualquer tipo de sistema de arquivos só é viável porque o próprio kernel implementa uma camada de abstração em torno de sua interface de sistema de arquivos de baixo nível. Esta camada de abstração permite que o Linux suporte diferentes sistemas de arquivos, mesmo que eles diferem muito em recursos ou comportamentos suportados.

O resultado é uma camada geral de abstração que permite que o kernel suporte muitos tipos de sistemas de arquivos de forma fácil e limpa. Os sistemas de arquivos são programados para fornecer as interfaces abstratas e estruturas de dados que o VFS espera; por sua vez; o kernel funciona facilmente com qualquer sistema de arquivos e a interface de espaço de usuário exportada funciona perfeitamente em qualquer sistema de arquivos.