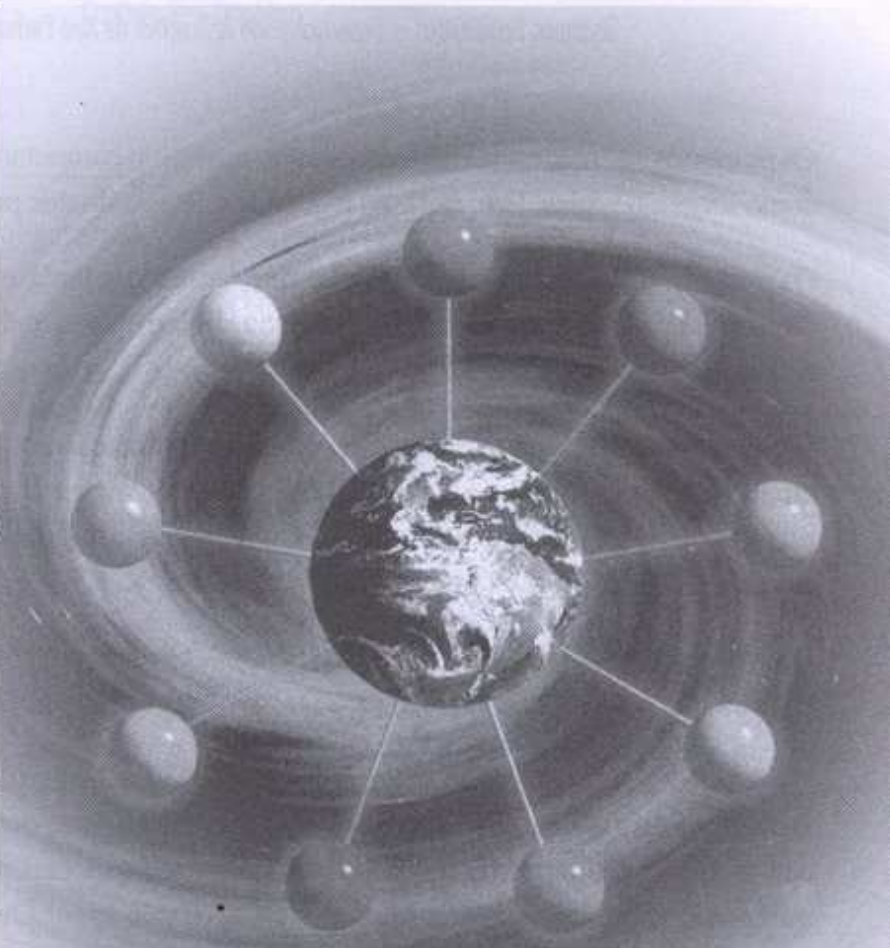
CAPÍTULO



3

Processos Concorrentes

Os processos são programas carregados na memória do computador e podem ser classificados como leves e pesados. Este capítulo é dedicado ao estudo dos processos concorrentes.

Os processadores executam instruções que representam algoritmos. Um processo é uma entidade abstrata, formada pelos recursos de hardware, pela execução das instruções referentes a um algoritmo.

Um algoritmo escrito em uma linguagem de programação pode ser compilado, resultando em um código objeto que é carregado na memória para execução ou pode ser interpretado por um outro programa e executado. Os processos compilados podem ser classificados em dois tipos: processos pesados e processos leves.

Os processos pesados são os tradicionais. Eles possuem um conjunto de operações iniciais básicas que começam a sua execução. Estas operações iniciais são o que chamamos de thread. A função "*void main*" de um programa escrito em C é um thread inicial.

Para entendermos os processos, precisamos saber o que ocorre quando o computador com o Linux é ligado. Inicialmente, um software especial residente na placa-mãe dos microcomputadores, conhecido como BIOS (Basic Input Output System), faz os testes no hardware e identifica os dispositivos conectados ao micro. Após, ela imediatamente procura no setor de boot do disco rígido o gerenciador de boot. No Linux gerenciadores mais comuns são o LILO e o Grub. Eles são responsáveis pela carga do Kernel na memória. Após a carga do Kernel, este inicia um processo especial chamado *init*. Este processo é o pai de todos os processos e responsável pelo restante da carga do boot do Linux.

Depois da carga do boot, o *init* chama outro programa especial chamado *getty*, que é responsável pela autenticação dos usuários e pelo início do processo de *shell*.

"O maior empresário de todos os tempos foi Tutankamon, que construiu as pirâmides há 5 mil anos e elas até hoje produzem riqueza."

Peter Drucker, pai da administração moderna

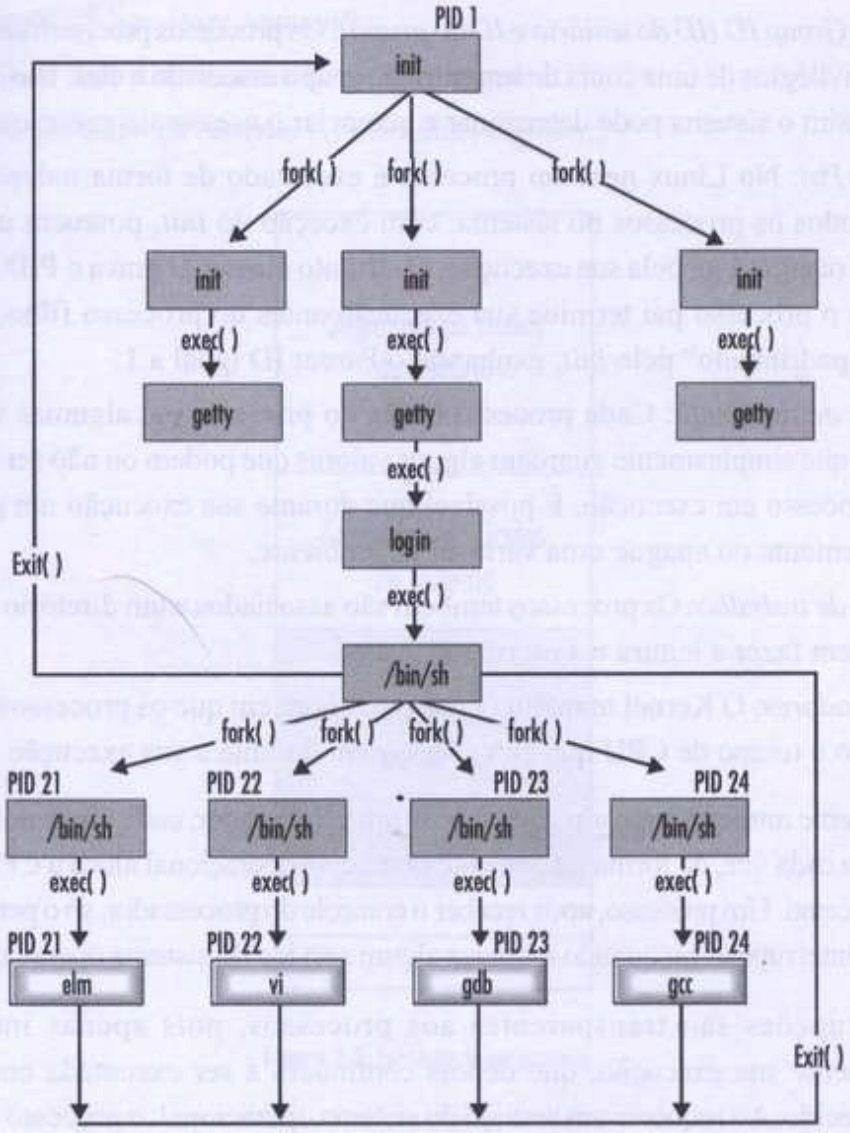


Figura 3.1: Carga do sistema Linux.

É importante que você entenda que cada programa é pelo menos um processo e que cada processo possui alguns atributos, tais como:

- *Process ID (PID) ou identificação do processo*: Cada processo possui um número de identificação único. O primeiro processo init sempre terá o PID 1 e para o restante dos processos este número é incrementado à medida que novos processos são executados.

- *User ID e Group ID (ID do usuário e ID do grupo)*: Os processos precisam ser executados com os privilégios de uma conta de usuário e do grupo associado a eles. Isto é importante porque assim o sistema pode determinar e gerenciar o acesso aos recursos.
- *Processo Pai*: No Linux nenhum processo é executado de forma independente dos outros. Todos os processos no sistema, com exceção do *init*, possuem um processo pai, que é responsável pela sua execução. O atributo *ParentID* grava o PID do processo pai. Caso o processo pai termine sua execução antes do processo filho, o processo filho é "apadrinhado" pelo *init*, ganhando o Parent ID igual a 1.
- *Variáveis de ambiente*: Cada processo herda do processo pai algumas variáveis de ambiente que simplesmente guardam alguns valores que podem ou não ser importantes para o processo em execução. É possível que durante sua execução um processo altere, incremente ou apague uma variável de ambiente.
- *Diretório de trabalho*: Os processos também são associados a um diretório de trabalho, onde podem fazer a leitura e a escrita no disco.
- *Temporizadores*: O Kernel mantém registros da hora em que os processos são criados bem como o tempo de CPU que eles consomem durante a sua execução.

Em um ambiente multiprogramado com apenas um processador, cada processo é executado aos poucos de cada vez, de forma intercalada. O sistema operacional aloca a CPU um pouco para cada processo. Um processo, após receber o controle do processador, só o perderá quando ocorrer uma interrupção ou quando requerer algum serviço do sistema operacional.

Estas interrupções são transparentes aos processos, pois apenas interrompem temporariamente sua execução, que depois continuará a ser executada como se nada tivesse acontecido. Ao requerer um serviço do sistema operacional, o processo é bloqueado até que o serviço requerido ao sistema operacional seja satisfeito.

Lembrando o capítulo anterior, os processos, sejam pesados ou leves, podem alterar entre três estados de execução diferentes. Um processo está no "estado de execução" quando suas instruções estão sendo executadas pelo processador. Ele entra no "estado pronto" quando possui condições para ser executado e está esperando pelo processador. E finalmente está em "estado bloqueado" quando está aguardando alguma condição, por exemplo, a espera de uma operação de entrada e saída.

Além de um estado de execução, um processo ocupa uma área de memória formada basicamente por 3 partes:

- Segmento de código (*text segment*)
- Segmento de dados de usuário (*user data segment*)
- Segmento de dados de sistema (*system data segment*)

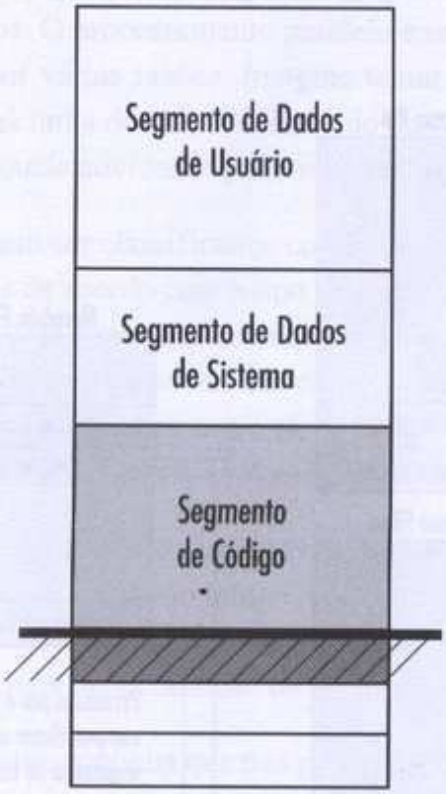


Figura 3.2: Estrutura de um processo.

O segmento de código contém as instruções de máquina geradas na compilação do programa. O segmento de dados de usuário contém as variáveis utilizadas pelo programa. É conveniente que as instruções estejam separadas dos dados, pois isso possibilita o compartilhamento de código por vários programas em execução. Neste caso diz-se que o procedimento é *reentrante* ou *puro*.

Se cada programa em execução possui uma pilha própria, então os dados podem ser alocados na própria pilha do programa. Além das instruções e dados, cada programa em execução possui uma área de memória correspondente para armazenar os valores dos

registradores da CPU para quando não estiver sendo executado por algum motivo. Essa área de memória é conhecida como registro descritor ou segmento de dados do sistema.

Um mesmo programa pode ser ativado mais de uma vez, dando origem a vários processos. Nesse caso o segmento de código (*text segment*) é compartilhado entre eles, conforme mostrado na figura.

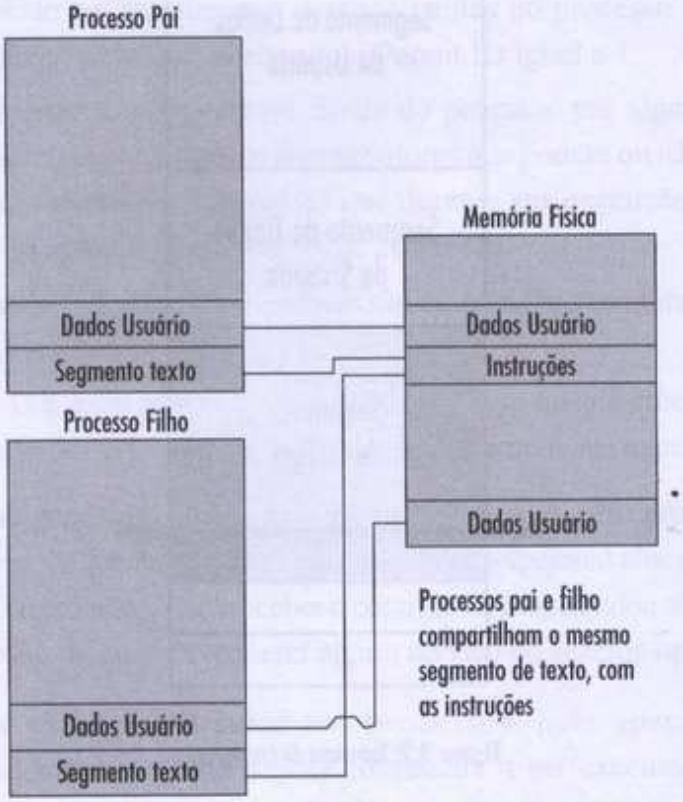


Figura 3.3: Compartilhamento do segmento de código entre os processos.

Concluindo, um processo é uma entidade completamente definida por si só, cujas instruções executadas se desenvolvem no tempo, em uma ordem que é função exclusiva dos valores iniciais de suas variáveis e dos dados lidos durante a execução.

Em um sistema com mais de um processador o sistema operacional passa a dispor de mais CPUs para alocar os processos. O resultado é a execução simultânea real de vários processos.

Todos os processos que existam ao mesmo tempo são concorrentes. Eles podem funcionar completamente independentes uns dos outros, ou ocasionalmente necessitar de sincronização e cooperação.

Dizemos ainda que, se certas operações podem ser logicamente executadas em paralelo, temos os processos paralelos. O processamento paralelo é um assunto interessante e ao mesmo tempo complexo por várias razões. Imagine tentar fazer uma pessoa ler dois livros ao mesmo tempo: uma linha de um, uma linha do outro, e assim por diante. Desta forma, é difícil determinar quais atividades podem ou não ser executadas em paralelo.

Os processos paralelos podem ser classificados como concorrentes ou assíncronos. Os primeiros pode ser divididos de acordo com o tipo de interação existente entre eles. São disjuntos quando trabalham com conjuntos distintos de dados, ou interativos quando têm acesso a dados comuns. Estes últimos podem ser competitivos se brigarem por recursos e cooperativos se trocarem informações entre si. Já os processos assíncronos podem ocasionalmente interagir um com o outro, às vezes de forma complexa.

Quando estamos lidando com processos interativos, a ordem das operações sobre as variáveis compartilhadas podem variar no tempo, uma vez que as velocidades relativas dos processos dependem de fatores externos. Toda vez que a ordem de execução causa interferência no resultado temos uma condição de corrida – *race condition*.

Para entendermos este conceito, suponha que três processos compartilhem uma variável *A*. Dois deles fazem operações de modificação e um imprime o valor da variável. Suponha também que a ordem em que estas operações são feitas é importante. Se a variável *A* for modificada simultaneamente pelos dois processos, o valor impresso irá depender de quando as operações de escrita forem realizadas. Para colocar em termos concretos, imagine se $A = 11$, e a seqüência de operações seja a seguinte:

```
Processo 1 -=> A = A + 5 ;
Processo 2 -=> A = A + 1 ;
Processo 3 -=> Imprimir A ;
```

Na ordem acima, processo 3 irá imprimir o valor 17. Se a ordem for diferente, teremos outro valor para *A*.

Neste exemplo, podemos ver que os processos em execução compartilham o acesso a uma mesma variável. Eles são concorrentes interativos e estão em condição de corrida.

As condições de corrida precisam ser evitadas para garantir que o resultado de um processamento não varie entre uma execução e outra. Condições de corrida resultam em computações paralelas errôneas, pois cada vez que o programa for executado com os mesmos dados poderão ser obtidos resultados diferentes.

A programação paralela exige mecanismos de sincronização entre processos, e por isso sua codificação e depuração são mais complexas, mas intelectualmente atraentes.

Imagine um sistema que deva atender requisições de serviço que ocorram de forma imprevisível e aleatória. Ele poderá ser construído para permitir que cada solicitação seja realizada por um de seus processos paralelos, gerando economia dos recursos computacionais.

A programação paralela exige do sistema operacional o suporte necessário para a criação e gerência de processos concorrentes, mecanismos de sincronização e comunicação entre processos, acesso aos dispositivos locais e remotos, e o controle de acesso a recursos compartilhados.

Veja a seguir as chamadas de sistema para a criação de processos concorrentes.

Identificando um Processo

O identificador de cada processo PID é fornecido pelo sistema operacional. A linguagem C tem diferentes primitivas que permitem conhecer este número. Elas serão freqüentemente utilizadas nos exemplos para identificarmos os processos pai e filho.

Essas primitivas estão declaradas no cabeçalho *unistd.h*.

pid_t getpid( )

A função *getpid( )* retorna o número PID do processo em execução.

pid_t getppid( )

A função *getppid( )* retorna a identificação do processo pai, que criou o processo em execução, através de outras diretivas como o *fork( )*, *system( )* e *exec( )*.

pid_t getpgrp( )

Esta função retorna o ID do grupo do processo. Os grupos de processos são usados para distribuição de sinais entre os processos relacionados. Veremos o conceito de sinais adiante.

Exemplo:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   printf("Eu sou o processo %d. Meu pai é o %d. O ID do grupo de
processo é o %d.\n",getpid(), getppid(), getpgrp());
   exit(0);
};
```

Para compilar este programa salvo como "*meupid.c*":

```
# gcc meupid.c -o meupid
```

O resultado da execução será:

```
# ./meupid
```

Eu sou o processo 21335. Meu pai é o 5826. O ID do grupo de processo é o 21335.

A Primitiva *exec( )*

A primitiva *exec( )* é representada pelas funções *execl( )*, *execle( )*, *execlp( )*, *execv( )*, *execve( )* e *execvp( )*. Cada uma das funções desta família trocam a imagem do processo em execução pela imagem de outro processo.

O arquivo que será chamado por estas funções precisa ser um arquivo binário ou script para ser interpretado por um outro processo. De qualquer forma, ele precisa ter permissões de execução.

Estas funções não retornam nenhum valor que indique seu sucesso uma vez que o processo em execução é inteiramente trocado pelo processo novo. No caso de falha será retornado -1.

NOTA

Um script para ser interpretado no Linux precisa necessariamente iniciar com a indicação de qual programa será responsável por sua interpretação. Para fazer isso, sua primeira linha deve começar com "#!/caminho/programa". Por exemplo: #!/usr/bin/perl.

As funções desta família diferem basicamente na passagem de parâmetros, mas fazem a mesma coisa. Elas são declaradas no cabeçalho *unistd.h*.

int execv (const char *arquivo, char *const argv[ ])

A função *execv( )* executa o programa informado em *arquivo* e os seus possíveis parâmetros são passados pela string *argv*, trocando a imagem do processo na memória.

O array de string dos argumentos deve necessariamente terminar em um ponteiro nulo *(char *)0*. Por convenção, o primeiro elemento deve ser o nome do programa chamado. As variáveis ambientais do novo processo são herdadas do processo em execução.

Observe o seguinte exemplo:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    char *cmd[] = { "cat", "/etc/passwd", (char *)0 };
    printf("Vou chamar o programa cat para ler o conteúdo de /etc/
passwd.\n");
    execv("/bin/cat",cmd);
    printf("Esta mensagem não será impressa.\n") ;
    exit(0);
}
```

Para compilar o programa, salvo com o nome *"execv.c"*:

```
# gcc execv.c -o execv
```

O resultado será o seguinte:

```
Vou chamar o programa cat para ler o conteúdo de /etc/passwd.
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
mail:x:8:12:Mailer daemon:/var/spool/clientmqueue:/bin/false
games:x:12:100:Games account:/var/games:/bin/bash
at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
wwwrun:x:30:8:WWW daemon apache:/var/lib/wwwrun:/bin/false
squid:x:31:65534:WWW-proxy squid:/var/cache/squid:/bin/false
ftp:x:40:49:FTP account:/srv/ftp:/bin/bash
```

A mensagem do *printf* depois do *execv* não será impressa, pois o processo será substituído pelo programa *cat*.

int execl (const char *arquivo, const char *args, ... 0)

Esta função é similar ao *execv*, mas os argumentos devem ser passados individualmente.

Observe o seguinte exemplo:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   printf("Vou chamar o programa head para ler o cabeçalho de /etc/
passwd e /etc/group.\n");
   execl("/usr/bin/head","head", "/etc/passwd" , "/etc/group", (char *)0);
   printf("Esta mensagem não será impressa.\n") ;
   exit(0);
}
```

Para compilar o programa, salvo com o nome *"execl.c"*:

```
# gcc execl.c -o execl
```

O resultado da execução será o seguinte:

```
Vou chamar o programa head para ler o conteúdo de /etc/passwd e /etc/
group.
==> /etc/passwd <==
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
```

```
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
mail:x:8:12:Mailer daemon:/var/spool/clientmqueue:/bin/false
==> /etc/group <==
root:x:0:
bin:x:1:daemon
daemon:x:2:
sys:x:3:
tty:x:5:
```

int execve (const char *arquivo, char *const argv[ ], char *const env[ ])

Esta função também é similar ao *execv*. Ela permite passar argumentos e variáveis ambientais para o programa chamado. O argumento *argv* define os argumentos e *env* define as variáveis ambientais. Ambas são um array de string.

As variáveis ambientais são mantidas pelo *bash* e contêm alguma informação importante para a execução do shell ou outro programa. Deverão estar no formato *NOME=VALOR*.

Estas variáveis são carregadas no início da execução do bash e também podem ser configuradas manualmente a qualquer momento. O conteúdo de qualquer variável do shell poderá ser visualizado com o comando *echo* sucedido pelo símbolo $ mais o nome da variável:

```
$ echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin
```

Uma lista completa das variáveis do shell poderá ser obtida com o comando:

```
$ set
```

Para alterar ou criar uma nova variável do shell:

```
$ LIVRO="Sistemas Distribuídos"
$ export LIVRO
```

Preferencialmente as variáveis devem ser declaradas em caixa alta e sem espaços entre o nome da variável, o símbolo "=" e o seu conteúdo. Se o conteúdo for alfanumérico, é desejável que ele esteja entre aspas simples ou duplas. Depois de criar uma variável do shell, é preciso exportá-la para o ambiente com o comando *export*. Quando uma variável é exportada para o ambiente, ela fica disponível para todos os processos filhos do shell (todos os programas e aplicações que foram executados no bash).

A função *execve* cria uma variável ambiental recebida como parâmetro para que ela esteja disponível para processo requisitado. Ao término deste, a variável deixa de existir. As variáveis ambientais já disponíveis no shell ficam disponíveis para todos os seus processos filhos.

Veja o exemplo:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    char *cmd[] = { "echo", "Olá", (char *)0 };
    char *env[] = { "LIVRO=Sistemas Distribuidos", (char *)0 };
    printf("Vou chamar o programa echo para escrever Olá.\n");
    execve("/bin/echo", cmd, env);
    printf("Esta mensagem não será impressa.\n") ;
    exit(0);
}
```

Para compilar o programa, salvo com o nome *"execve.c"*:

```
# gcc execve.c -o execve
```

O resultado da execução será o seguinte:

```
Vou chamar o programa echo escrever Olá.
```

int execle (const char *arquivo, const char *arg0, char *const env[ ], ...)

Esta função é parecida com a anterior e com a função *execl*. Ela permite que mais de um argumento seja passado explicitamente, como também mais de uma variável ambiental. Elas devem ser separadas pelo valor 0, indicando uma terminação NULA. Por exemplo:

```
execle("/bin/programaqualquer", "parâmetro1", "parâmetro2",0,
"VARIAVEL1=1234", "VARIAVEL2=5678", (char *)0)
```

Observe uma forma elegante de utilizar o *execle*:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
```

```
    int ret;
    char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
    printf("Vou chamar o programa ls para ler o conteúdo do diretório
corrente.\n");
    execle ("/bin/ls", "ls", "-l", (char *)0, env);
    printf("Esta mensagem não será impressa.\n") ;
    exit(0);
}
```

Para compilar o programa salvo como "*exele.c*"

```
# gcc execle.c -o execle
```

O resultado da execução será:

```
# ./execle
Vou chamar o programa ls para ler o conteúdo do diretório corrente.
total 80
-rwxr-xr-x    1 root     root         8640 Jan 12 00:04 execl
-rw-r--r--    1 root     root          281 Jan 12 00:04 execl.c
-rwxr-xr-x    1 root     root         8640 Jan 11 23:58 execv
-rw-r--r--    1 root     root          261 Jan 11 23:58 execv.c
-rwxr-xr-x    1 root     root         8642 Jan 12 00:10 execve
```

int execvp (const char *arquivo, char *const argv[ ])

Esta função opera basicamente como o *execv*, mas permite que o programa passado como argumento seja procurado nos diretórios especificados na variável ambiental *$PATH*, se não for passado seu caminho completo. Ela é útil para chamar programas que são utilitários do sistema e podem variar de acordo com a distribuição do Linux ou Unix.

Veja o exemplo a seguir:

```
#include <unistd.h>
#include <stdio.h>
int main()
{
int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
printf("Imprimindo o conteúdo do diretório corrente\n\n");
ret = execvp ("ls", cmd);
}
```

Para compilar o programa salvo como "*exevp.c*"

```
# gcc execvp.c -o execvp
```

O resultado da execução será:

```
# ./execvp
Imprimindo o conteúdo do diretório corrente
total 128
-rwxr-xr-x    1 root     root         8639 Jan 12 00:13 exec
-rw-r--r--    1 root     root          260 Jan 12 00:13 exec.c
-rwxr-xr-x    1 root     root         8640 Jan 12 00:19 execl
```

Na função *execv* o caminho do programa não precisa estar completo, desde que esteja nos diretórios de busca.

int execlp (const char *arquivo, const char *arg0, ...)

Esta função funciona como a *execl*, aceitando os argumentos separadamente. A diferença é que ela também faz a procura do programa em *$PATH* se não for fornecido seu caminho completo.

Cada sistema operacional tem um tamanho máximo para os argumentos e variáveis ambientais. Consulte os limites do seu sistema Linux através do comando:

```
# grep ARG_MAX /usr/include/linux/limits.h
#define ARG_MAX       131072    /* # bytes of args + environ for
exec() */
```

Os descritores de arquivos abertos do processo em execução permanecem abertos para o processo chamado, exceto se o flag FD_CLOEXEC – fechar ao terminar – estiver habilitado. Todos os atributos dos descritores ficam intocados.

As funções da família *exec( )* destroem os buffers na região do usuário do processo anterior podendo haver perda de informações. Existem duas alternativas para esvaziar os buffers antes da chamada *exec( )*: a função *fflush( )* e o caractere de nova linha "*\n*".

Observe o exemplo a seguir:

```
#include <stdio.h>
#include <unistd.h>

int main()
```

```
{
    char *cmd[] = {"ls","-l", (char *)0 };
    printf("Imprimindo o conteúdo do diretório local") ;
    fflush(stdout) ;
    execv("/bin/ls",cmd) ;
    exit(0);
}
```

Para compilar o programa salvo como "*fflush.c*":

```
# gcc fflush.c -o fflush.c
```

A função *fflush( )* esvazia o buffer para a saída padrão. Experimente compilar e executar esse programa comentando a linha fflush(stdout).

Para resumir, observe a tabela com as funções *exec( )*:

Tabela 3.1: Funções da família *exec( )*

Função	Descrição	Lembrete
execv( )	Os parâmetros devem ser passados como uma única variável string.	*execV* – uma Variável.
execl( )	Os parâmetros podem ser passados em várias variáveis.	*execL* – uma Lista de variáveis.
execve( )	Os parâmetros devem ser passados como uma única variável string. Aceita variáveis ambientais.	*execVE* – uma Variável + Environment - ambiente em inglês.
execle( )	Os parâmetros podem ser passados em várias variáveis. Aceita variáveis ambientais.	*execLE* – uma Lista de variáveis + uma lista de Environment.
execvp( )	Os parâmetros devem ser passados como uma única variável string. Procura pelo processo no $PATH.	*execVP* – uma Variável + procura no $PATH.
execlp( )	Os parâmetros podem ser passados em várias variáveis.	*execLP* – uma Lista de variáveis + procura no $PATH.

Como o código do processo que chama uma função *exec( )* será sempre destruído, é interessante que sua utilização esteja associada com a primitiva *fork( )*.

A Primitiva *system( )*

A primitiva *system( )* executa um programa especificado como parâmetro e retorna ao processo original quando este termina. O processo original não é substituído, mas congelado.

Observe o seguinte exemplo:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
printf("Vou chamar o programa cat para ler o conteúdo de /etc/
issue.\n");
system("/bin/cat /etc/issue");
printf("Esta mensagem será impressa.\n") ;
exit(0);
}
```

Para compilar este programa, salvo como *"system.c"*:

```
# gcc system.c -o system
```

Para executar:

```
# ./system
Vou chamar o programa cat para ler o conteúdo de /etc/issue.
Welcome to SuSE Linux 9.0 (i586) - Kernel \r (\l).
Esta mensagem será impressa.
```

Após a execução do comando *"/etc/cat"*, o processo original é descongelado, sua próxima instrução será processada.

A função *system( )* recebe apenas um único valor, onde o comando e seus parâmetros devem ser passados como uma única variável string.

```
int system(const char *string);
```

"Por que cometer erros antigos se há tantos erros novos a escolher?"

Bertrand Russel, filósofo inglês

Esta função retorna -1 em caso de erro. Se for bem-sucedida, a função retorna o código de saída do programa chamado. Este código poderá ser lido através da macro *WEXITSTATUS(status)* que será explicada mais adiante.

IMPORTANTE

Nunca use as funções system( ), execvp( ) ou execlp( ) para executar um processo com os bits de suid ou sgid porque algumas variáveis ambientais podem ser usadas para subverter a integridade do sistema. Nestes casos, prefira usar as funções execv( ), execl( ), execve( ) ou execle( ).

As Primitivas *fork( )* e *wait( )*

As primitivas *fork( )* e *wait( )* foram as primeiras notações de linguagem para especificar concorrência. O *fork* inicia um novo processo que compartilha o espaço de endereçamento do processo criador. Ambos os processos, aquele que efetuou a chamada *fork* e o processo criado, continuam a execução em paralelo. Esta função retorna um valor que pode ser utilizado no controle de execução.

Estas primitivas estão declaradas no cabeçalho *unistd.h*. A figura a seguir ilustra a operação *fork*.

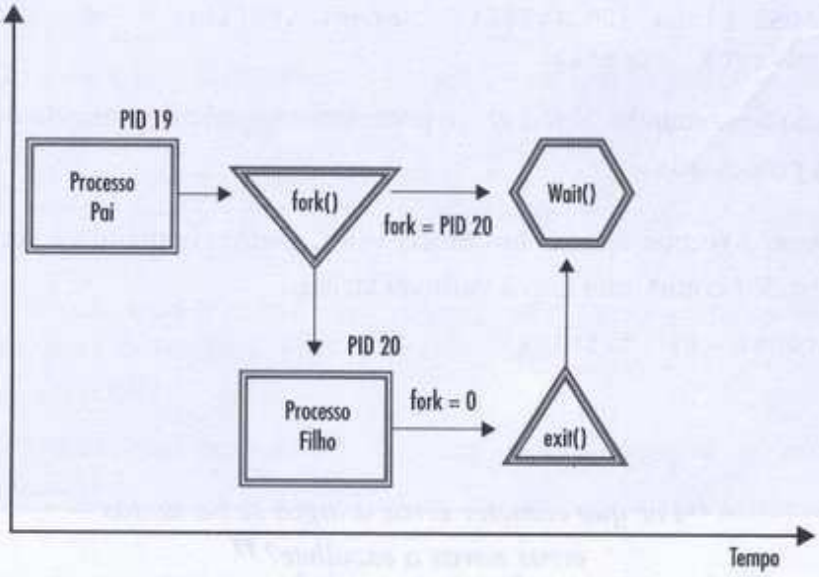


Figura 3.4: Criação de um processo filho através do fork.

Nesta oportunidade, o processo executa a operação *fork( )*, criando o processo filho B, dividindo o processamento em dois. Parte será executada pelo processo pai e parte pelo processo filho.

As características do *fork* no Linux/Unix são as seguintes:

- Ele duplica o processo que executa a chamada e os dois executam o mesmo código;
- Cada processo tem seu próprio espaço de endereçamento, com cópia de todas as variáveis, que são independentes em relação às variáveis do outro processo;
- Ambos executam a instrução seguinte à chamada de sistema;
- O retorno da chamada *fork* para o processo pai contém a identificação do Process ID (PID) do processo filho criado;
- O retorno da chamada *fork* para o processo filho é zero;
- Pode-se selecionar o trecho de código que será executado pelos processos com a instrução condicional *if*;
- A sincronização entre os processos pai e filho é feita com a primitiva *wait( )*, que bloqueia o processo que a executa até que um processo filho termine.

Exemplo de programa com *fork* no Unix/Linux:

```
#include <unistd.h>
int main() {
    int id;
    id = fork();
    if (id != 0)
    {
       printf("Eu sou o pai e espero pelo meu filho %d\n",id);
       wait(0);
       printf("Meu filho acabou de terminar… Vou terminar também!\n");
    }
       else
    {
       printf("Eu sou o filho %d e espero 10 segundos.\n",getpid());
       sleep(10);
       printf("Já esperei e vou embora...\n");
    };
};
```

Para compilar este programa, salvo com o nome *fork_simples.c*:

```
# gcc fork_simples.c -o fork_simples
```

Como resultado da execução:

```
# ./fork_simples
Eu sou o filho 8484 e espero 10 segundos.
Eu sou o pai e espero pelo meu filho 8484
Já esperei e vou embora...
Meu filho acabou de terminar. Vou terminar também!
```

Neste programa um processo irá chamar a primitiva *fork( )*. Neste momento, o processo será duplicado, criando um novo processo filho. Ambos terão o mesmo código, mas com o seu próprio espaço de endereçamento e cópia de todas as variáveis, de forma independente em relação às variáveis do outro processo. Ambos irão executar a instrução seguinte à chamada de sistema. A função *fork* irá retornar um número que será carregado na variável *id*.

Para o processo pai, que executou o *fork*, o valor na variável *id* será o número de identificação PID que o sistema operacional deu ao processo filho. Para o processo filho o valor da variável *id* será sempre zero.

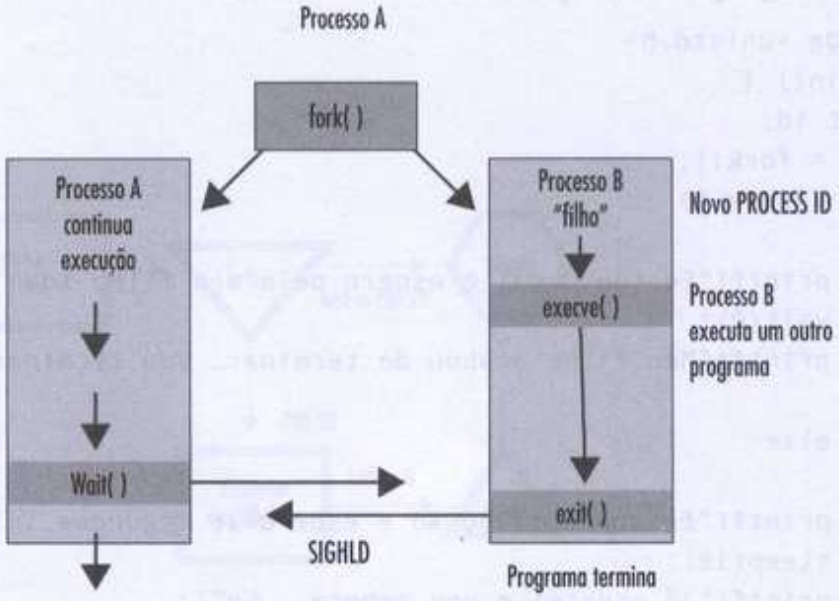


Figura 3.5: Exemplificação do fork( ) e wait( ).

Assim, para o processo pai o teste *if (id != 0)* será verdadeiro e serão executadas as instruções *printf* e *wait (0)*. Esta última é uma primitiva do sistema que bloqueia a execução do processo que a invocou até que o processo filho criado pelo *fork* termine.

Para o processo filho o teste do *if* será falso e serão executadas as instruções *printf* e *sleep(10)*. Assim o processo filho irá esperar 10 segundos e terminar. O seu término desbloqueará o processo pai enviando um sinal para a primitiva *wait( )*, que irá liberar a instrução *printf* e logo depois também terminará.

Para demonstrar ainda mais a criação de um novo processo, o seguinte exemplo foi elaborado para deixar o processo filho executando indefinidamente.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
int pid;
printf("Meu número de processo é %d. Vou criar um processo
filho.\n",getpid());
pid=fork(); /* Chamada da primitiva fork */
if(pid==0) /* Aqui começam as instruções do processo filho. */
{
    printf("\t\tProcesso filho criado com o PID número %d.\n",getpid());
    printf("\t\tVou ficar executando indefinidamente.\n");
    for(;;); /* Este processo ficará em loop */
}
else /* Aqui começam as instruções do processo pai */
{
    sleep(5);
    printf("O processo pai termina e deixa o processo filho orfão.\n");
    printf("Veja se o processo filho continua rodando com o comando\n");
    printf("ps\n");
}
exit(0);
}
```

Neste exemplo o processo pai irá criar um processo filho. Este novo processo irá executar indefinidamente um *loop*. O processo pai irá terminar sua execução normalmente.

Para compilar este programa salvo com o nome "*fork_loop.c*":

```
# gcc fork_loop.c -o fork_loop
```

O resultado da execução será:

```
# ./fork_loop
Meu número de processo é 5046. Vou criar um processo filho.
                Processo filho criado com o PID número 5047.
                Vou ficar executando indefinidamente.
O processo pai termina e deixa o processo filho orfão.
Veja se o processo filho continua rodando com o comando
ps
```

Agora, para podermos ver o processo filho ainda em execução, devemos rodar o programa *ps*:

```
# ps
PID TTY          TIME     CMD
22098 pts/49     00:00:00 bash
5047 pts/49      00:00:10 fork_loop
5094 pts/49      00:00:00 ps
```

Neste exemplo, o processo chamado *fork_loop* está em execução com o PID 5047. Para terminar este programa, deve-se utilizar o comando *kill* passando como parâmetro o PID do processo.

```
# kill 5047
```

A Herança Entre os Processos Pai e Filho

É importante saber ainda que a cópia do segmento de dados do processo filho pode diferir do segmento do processo pai, como na identificação de processo, tempo de execução e outros atributos. No entanto, os processos filhos herdam uma duplicata de todos os descritores dos arquivos abertos do processo pai, de forma que os ponteiros para os arquivos associados são divididos. Neste caso, se o processo filho movimenta o ponteiro dentro de um arquivo, o processo pai irá fazer a próxima movimentação na nova posição.

Veja o exemplo a seguir, um pouco mais complexo:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>
int main()
{
int pid; /* variável para pegar o retorno do fork */
int fd; /* descritor de arquivo associado ao arquivo aberto */
char*pidnum; /* variável que vai receber o valor de getpid */
int r; /* retorno da leitura do arquivo meupid*/
int i; /* apenas um contador */
char c; /* variável que recebe o conteúdo do arquivo */
     printf("Meu número de PID é %d\n",getpid());
     printf("Vou gravar meu número de PID no arquivo meupid.\n");
     if((fd=open("meupid",O_CREAT|O_RDWR|O_TRUNC,S_IRWXU))==-1)
     /* Cria um arquivo com o nome meupid e permissões de leitura e
gravação */
     {
          printf("Não consegui criar o arquivo meupid.");
          exit(-1) ;
     }
     sprintf(pidnum,"%d surpresa",getpid()); /* Imprime o valor de
getpid na variável pidnum */
     if(write(fd,pidnum,15)==-1)
     {
          printf("Não consegui escrever no arquivo.");
          exit(-1) ;
     }
     printf("Já escrevi o número. Fechando o arquivo.\n");
     close(fd);
     printf("Vou abrir novamente o arquivo para leitura.\n");
     if((fd=open("meupid",O_RDONLY,S_IRWXU))==-1)
     {
          printf("Não foi possivel abrir o arquivo.");
          exit(-1) ;
     }
     printf("Vou criar um processo filho agora.\n");
     pid=fork();
     if(pid==-1) /* tratamento de erro */
     {
          perror("Não foi possivel criar um processo filho.");
          exit(-1) ;
```

```
    }
    else if(pid==0) /* aqui começam as instruções do processo filho */
    {
        printf("\t\tSou o processo filho, meu PID é %d.\n",getpid());
        printf("\t\tVou ler o arquivo que já estava aberto pelo
processo pai antes\n");
        printf("\t\tda minha criação e que eu herdei.\n");
        for(i=1;i<=5;i++) /* Faz um loop 5x para ler o PID do arquivo
meupid */
        {
            if(read(fd,&c,1)==-1)
            {
                printf("Não consegui lêr o arquivo.");
                exit(-1);
            }
            printf("\t\tO número que foi lido no arquivo é
%c\n",c);
        }
        printf("\t\tFechando o arquivo e terminando minha
execução...\n") ;
        close(fd) ;
        exit(1) ;
    }
    else /* aqui começam as instruções do processo pai */
    {
        wait(); /* O processo pai espera até que o processo filho
termine. */
        printf("O processo filho criado tinha o PID número
%d\n",pid) ;
        printf("Vou ler o arquivo meupid.\n") ;
        while((r=read(fd,&c,1))!=0)
        {
            if(r==-1)
            {
                printf("impossivel de ler o arquivo.");
                exit(-1) ;
            }
            printf("Consegui lêr -=> %c\n",c);
        }
        printf("Meu filho leu o número do PID no arquivo e mexeu no meu
ponteiro.\n") ;
```

```
            printf("Terminando minha execução.\n") ;
            close(fd) ;
        }
        exit(0);
}
```

Para compilar este programa salvo com o nome "*fork_ponteiros.c*":

```
# gcc fork_ponteiros.c -o fork_ponteiros
```

Como resultado da execução:

```
# ./fork_ponteiros
Meu número de PID é 20052
Vou gravar meu número de PID no arquivo meupid.
Já escrevi o número. Fechando o arquivo.
Vou abrir novamente o arquivo para leitura.
Vou criar um processo filho agora.
      Sou o processo filho, meu PID é 20053.
      Vou ler o arquivo que já estava aberto pelo processo pai
      antes da minha criação e que eu herdei.
      O número que foi lido no arquivo é 2
      O número que foi lido no arquivo é 0
      O número que foi lido no arquivo é 0
      O número que foi lido no arquivo é 5
      O número que foi lido no arquivo é 2
      Fechando o arquivo e terminando minha execução...
O processo filho criado tinha o PID número 20053
Vou ler o arquivo meupid.
Consegui ler -=>
Consegui ler -=> s
Consegui ler -=> u
Consegui ler -=> r
Consegui ler -=> p
Consegui ler -=> r
Consegui ler -=> e
Consegui ler -=> s
Consegui ler -=> a
Consegui ler -=>
Meu filho leu o número do PID no arquivo e mexeu no meu ponteiro.
Terminando minha execução.
```

Neste exemplo o processo pai cria um arquivo para escrita chamado *meupid* e escreve o seu número de identificação, mais a palavra "surpresa". Depois ele fecha o arquivo para escrita e abre para leitura. O processo pai então cria um processo filho com a diretiva *fork*. O processo filho recém-criado herda do pai os descritores do arquivo já abertos e faz a leitura de 5 caracteres do seu conteúdo, imprime na tela e termina sua execução. O pai enquanto isso fica esperando que o processo filho termine com a diretiva *wait*. Depois que o filho termina, o pai volta a executar e lê o arquivo *meupid*. Como o filho andou com o ponteiro deste arquivo, o processo pai consegue ler o restante da frase "surpresa".

As Primitivas *wait( )* e *waitpid( )*

A primitiva *wait* tem como objetivo suspender a execução do processo pai até que o processo filho termine sua execução. Esta instrução é especialmente útil, principalmente para evitar "race conditions".

Outra primitiva da mesma classe é a *waitpid( )*. Ela suspende a execução de um determinado processo até que o processo filho especificado pelo PID informado tenha terminado.

Elas estão descritas no cabeçalho *wait.h*.

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options)
```

Estas primitivas retornam o valor do PID do processo que foi terminado. Caso ocorra um erro inesperado, irão retornar -1.

A variável *status* contém o número do sinal de término do processo esperado. Este número é passado pelo processo através da primitiva *exit( )*. Veremos isto adiante.

A função *waitpid* precisa receber como argumento um número de PID de um processo filho ou os seguintes valores predeterminados:

- *< -1* - Um valor menor que 1 diz a função *waitpid* para suspender o processamento e esperar o término de qualquer processo filho cujo valor do seu ID de grupo de processo seja igual ao número absoluto do valor passado.
- *-1* – Diz a função para suspender o processamento até que qualquer processo filho termine. Este é o comportamento padrão.

- *0* – Um valor igual a zero diz a função para suspender o processamento até que um processo filho com o mesmo ID de grupo do processo pai termine.
- *> 0* - Um valor maior que zero diz a função para suspender o processamento até que o processo filho identificado pelo PID informado esteja terminado.

Veja o exemplo abaixo:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
int pid;
printf("\nMeu PID é %d e meu grupo ID é %d. Irei criar um processo
filho.\n",getpid(),getpgrp());
if (fork() == 0) /* Aqui começam as instruções do processo filho */
{
    printf("\t\tProcesso filho criado. Meu PID é %d e meu grupo é
%d..\n",getpid(),getpgrp());
    sleep(4);
    printf("\t\tVou sair com o código 7.\n\n");
    exit(7) ;
}
else  /* Aqui começam as instruções do processo pai */
{
    int ret1, status1;
    printf("Sou o processo pai. Estou esperando o processo filho.\n");
    ret1 = wait(&status1);
    printf("O processo filho que terminou foi o PID %d.\n",ret1);
    printf("Ele terminou com o parâmetro de exit() igual a
%d.\n",(status1>>8));
}
exit(0);
}
```

Neste exemplo o processo executa o *fork* e cria um processo filho. Este processo filho termina sua execução com a primitiva *exit(7)*; enquanto isso o pai suspende sua execução com a primitiva *wait( )* até que o filho termine.

A primitiva *wait* irá retornar para a variável *ret1* o valor do PID do processo filho que terminou. A variável *status1* irá conter o sinal de saída igual a 7 enviado pela função *exit* do processo. Este parâmetro irá conter nos seus 7 primeiros bits o número do sinal.

Para compilar este programa salvo com o nome "*wait_simples.c*":

```
# gcc wait_simples.c -o wait_simples
```

O resultado da execução será:

```
# ./wait_simples
Meu PID é 20698 e meu grupo ID é 20698. Irei criar um processo filho.
      Processo filho criado. Meu PID é 20699 e meu grupo é 20698..
Sou o processo pai. Estou esperando o processo filho.
      Vou sair com o código 7.
O processo filho que terminou foi o PID 20699.
Ele terminou com o parâmetro de exit( ) igual a 7.
```

Geralmente um processo termina em três situações:

- Quando acabar a execução de sua última instrução;
- Em circunstâncias especiais, quando ocorre um erro, como o endereço de um vetor fora dos limites;
- Um processo pode causar a morte de outros processos com primitivas especiais.

As primitivas que implementam o término de processos podem ser de dois tipos. A primitiva *exit( )* termina o processo normalmente e seus recursos são liberados pelo sistema operacional. As primitivas *abort(id)* e *kill(id)* são utilizadas pelo processo pai para terminarem forçosamente os processos filhos. O número de identificação (PID) do processo filho é necessário.

É possível, dentro de uma hierarquia de processos, que um determinado processo filho termine por algum motivo inesperado, se torne um processo zombie ou defunto (defunct). Os processos zombie não podem ser terminados com o comando kill, porque eles já não existem mais. É preciso terminar o processo pai.

Isso acontece porque cada processo criado recebe um lugar na tabela de processos do kernel. Quando ele termina, seu lugar na tabela do kernel recebe o resultado da sua execução. O resultado da execução é retido na tabela até alguém consultá-lo, quando, então, é removido da tabela.

Então o estado do processo é chamado de "zumbi" quando o mesmo termina e seu resultado ainda não foi retirado da tabela do kernel.

Outra possibilidade especial é quando um processo que tenha criado novos processos filhos que executam debaixo de sua hierarquia termine inesperadamente. Neste caso, os processos filhos perdem o seu processo pai e são adotados pelo processo *init* (PID 1) que é o pai de todos os processos. Observe a hierarquia entre os processos com o comando *pstree –c –p*.

Estas duas situações descritas não são normais e podem ser ocasionadas por bugs nos programas.

Veja o problema com zumbis no seguinte exemplo:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
int pid;
printf("Meu PID é %d e eu vou criar um processo filho.\n",getpid());
pid = fork();
if(pid == 0) /* Aqui começam as instruções do processo filho */
{
    printf("\t\tMeu pid é  %d e fui criado por
%d.\n",getpid(),getppid());
    printf("\t\tVerifique os processos em execução com o comando
ps.\n");
    sleep(10);
    printf("Processo filho terminando normalmente.\n");
    exit(0);
}
else /* Aqui começam as instruções do processo pai */
{
    for(;;);
}
}
```

Para compilar este programa, salvo com o nome *zombie.c*:

```
# gcc zombie.c -o zombie
```

Execute o programa em segundo plano.

```
# ./zombie &
```

Como resultado da execução:

```
Meu PID é 8482 e eu vou criar um processo filho.
      Meu pid é 8483 e fui criado por 8482.
      Verifique os processos em execução com o comando ps.
Processo filho terminando normalmente.
```

Durante a execução deste programa, você poderá observar com o comando *ps* que dois processos estão em execução. Faça isso antes que o filho termine.

```
# ps
  PID TTY          TIME     CMD
22098 pts/49       00:00:00 bash
 8482 pts/49       00:00:01 zombie
 8483 pts/49       00:00:00 zombie
 8484 pts/49       00:00:00 ps
```

Logo após a mensagem do processo filho "*Processo filho terminando normalmente*", execute novamente o comando *ps*.

```
PID TTY          TIME     CMD
22098 pts/49       00:00:00 bash
 8482 pts/49       00:00:35 zombie
 8483 pts/49       00:00:00 zombie <defunct>
 8652 pts/49       00:00:00 ps
```

Podemos ver claramente que o processo pai 8482 ficou em execução e que o processo filho 8483, mesmo depois de terminado, continuou na lista de processos, com a indicação *<defunct>*.

Para terminar definitivamente os processos, execute o comando *killall* informando o nome do processo.

```
# killall zombie
```

A Primitiva *exit( )*

A primitiva *exit( )* termina imediatamente o processo em questão e, se houver descritores de arquivos abertos, todos são fechados. Ela pertence ao cabeçalho *unistd.h*.

```
void exit(int status);
```

O *exit( )* não retorna nada. Apenas aceita como parâmetro um valor numérico que é enviado na forma de sinal para o processo pai. O processo pai pode por sua vez capturar este valor com a primitiva *wait( )*.

Foi convencionado que, quando a função *exit( )* envia o sinal zero – *exit(0)* – indica que um processo foi terminado normalmente. O -1 geralmente indica que ocorreu um erro de execução não esperado. Valores diferentes podem indicar erros esperados ou definidos.

O sinal enviado pelo *exit* pode ser recuperado pelo interpretador de comandos através de uma variável ambiental especial: o *$?*. Observe o próximo exemplo:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Meu PID é %d e vou terminar com o sinal 10.\n",getpid());
    sleep(5);
    printf("Saindo...\n");
    exit(10);
}
```

Para compilar este programa salvo com o nome "*wait_bash.c*":

```
# gcc wait_bash.c -o wait_bash
```

Para executá-lo:

```
# ./wait_bash
```

Como resultado:

```
Meu PID é 2716 e vou terminar com o sinal 10.
Saindo...
```

Para verificar código de saída do programa:

```
# echo $?
10
```

Se entrarmos com este comando novamente, o resultado será diferente, pois esta variável sempre lê o conteúdo do *exit* do último processo executado.

A linguagem C oferece algumas macros que facilitam o tratamento dos sinais enviados pelo *exit( )* e recebidos pela *wait( )*, como:

- *WEXITSTATUS (status)* – Retorna o valor dos primeiros sete bits do sinal recebido.
- *WIFEXITED (status)* – Retorna verdadeiro indicando término normal do processo.
- *WIFSIGNALED (status)* – Retorna verdadeiro se o término do programa for anormal.
- *WIFSTOPPED (status)* – Retorna verdadeiro se o processo tiver a sua execução congelada.

Estas funções e outras do gênero estão disponíveis no cabeçalho "*wait.h*".

Observe o exemplo:

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int main()
{
int pid;
int status;
printf("Meu PID é %d. Vou gerar um filho e ele vai terminar com o
sinal 5.\n",getpid());
pid = fork();
if (pid==0)
{
        printf("\t\tSou o filho, PID %d.\n",getpid());
        sleep(5);
        exit(5);
}
else
{
        printf("Estou esperando meu filho terminar.\n");
        wait(&status);
        if (WIFSIGNALED(status))
        {
                printf("Meu filho terminou ok.\n");
        }
        else
        {
                printf("Meu filho terminou com erro
%d.\n",WEXITSTATUS(status));
        };
}
printf("Saindo...\n");
exit(10);
}
```

Para compilar este programa salvo com "*wifsignaled.c*":

```
# gcc wifsignaled.c -o wifsignaled
```

O resultado da execução será o seguinte:

```
./wifsignaled
Meu PID é 10271. Vou gerar um filho e ele vai terminar com o sinal 5.
                Sou o filho, PID 10272.
Estou esperando meu filho terminar.
Meu filho terminou com erro 5.
Saindo...
```

Este programa cria um processo filho, que irá terminar com o código *exit(5)*. O processo pai fica esperando o término do processo filho com a função *wait( )*. O sinal de saída do processo filho é capturado pelo *wait* e gravado na variável *status*. A macro *WIFSIGNALED* testa se o processo terminou em condições normais lendo o conteúdo de *status*. Se o sinal recebido for diferente de 0, será impresso o sinal na tela com a função *WEXITSTATUS*. Esta função lê somente os primeiros 7 bits de *status*. Caso contrário será impresso que o término do processo filho foi ok.

Sinais

Cada processo no Linux fica à escuta de sinais. Estes sinais são utilizados pelo Kernel, por outros processos ou pelo usuário para avisar um determinado processo sobre algum evento em particular. O guru de UNIX W. Richard Stevens descreve os sinais como interrupções de software.

Tabela 3.2: Exemplos de sinais

Sinal	Valor Numérico	Ação
SIGHUP	1	Hang-Up ou desligamento. Este sinal é utilizado automaticamente quando você desconecta de uma sessão ou fecha um terminal. Ele também é utilizado por processos servidores para invocar a releitura do arquivo de configuração.

Continua

Continuação

Sinal	Valor Numérico	Ação
SIGINT	2	Interrompe o processo. Ele é enviado automaticamente quando abortamos um processo com as teclas ctrl-c.
SIGQUIT	3	Abandona o processo. Ele é enviado automaticamente quando abortamos um processo com as teclas ctrl-d.
SIGILL	4	Emitido quando uma instrução ilegal é detectada.
SIGIOT	6	Emitido quando existe um erro de entrada e saída (E/S).
SIGFPE	8	Emitido quando existe um erro de cálculo em ponto flutuante.
SIGKILL	9	Termina o processo incondicionalmente. Este tipo de sinal pode deixar arquivos abertos e de forma rápida e drástica. bases de dados corrompidas. Deve ser utilizado caso o processo pare de responder ou em uma emergência.
SIGEGV	11	Emitido quando existe uma violação na segmentação ao acessar um dado fora do endereçamento do processo.
SIGSYS	12	Emitido quando existem argumentos incorretos numa chamada de sistema.
SIGPIPE	13	Emitido quando existe um erro de escrita em um pipe.
SIGTERM	15	Termina o processo de forma elegante, possibilitando que ele feche arquivos e execute suas rotinas de fim de execução.
SIGTSTP	18	Termina a execução para continuar depois. Este sinal é enviado automaticamente quando utilizamos as teclas ctrl-z. É utilizado para colocar o processo em segundo plano ou enviado por um processo filho ao pai quando este termina sua execução.
SIGSTOP	19	Simplesmente termina o processo.

Quando um sinal é enviado para um processo, ele toma uma determinada ação dependendo do valor que este sinal carrega. Cada sinal no sistema tem um nome exclusivo e um valor numérico.

O Linux possui mais de 30 sinais definidos. A maioria é utilizada pelo kernel, para comunicação entre os processos e alguns pelos usuários. Seu entendimento é importante

para saber como o sistema interage com os processos em execução. O arquivo */usr/include/signal.h* contém a lista de sinais acessíveis ao usuário. O comando *man 7 signal* também fornece uma lista de sinais utilizados pelo sistema.

É importante saber que um sinal é normalmente tratado de duas maneiras diferentes:

- O processo recebe um sinal e faz o seu tratamento executando alguma ação.
- O processo recebe um sinal e o ignora.

Os sinais podem ser utilizados na programação distribuída para exercer a comunicação entre os processos concorrentes. A seguir veremos quais são as primitivas para o tratamento dos sinais.

Primitiva *kill( )*

A primitiva *kill( )* permite que qualquer sinal possa ser enviado para um determinado processo ou grupo de processos.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Esta primitiva está declarada no cabeçalho *signal.h*. Ela recebe dois parâmetros a saber:

- *sig* do tipo inteiro. Representa o número do sinal a ser enviado.
- *pid* do tipo *pid_t*. Representa um número de PID de um processo ou pode aceitar alguns com valores predeterminados. Estes são:
- *< -1* - Um valor menor que 1 diz à função para enviar o sinal para todos os processos cujo valor do ID de grupo seja igual ao número absoluto do valor passado.
- *-1* – Um valor igual a menos 1 diz à função para enviar um sinal a todos os processos, exceto o *init*.
- *0* – Um valor igual a zero diz à função para enviar um sinal a todos os processos que tenham o mesmo ID de grupo do processo emissor.
- *1* – Diz à função para enviar o sinal para todos os processos cujo dono seja igual ao dono do processo emissor. Se ele pertencer ao superusuário, o sinal é enviado para todos os processos de usuários.

- *> 0* – Um valor maior que zero diz à função para enviar o sinal para o processo identificado pelo número PID informado.

O *kill( )* retorna 0 se o sinal foi enviado e -1 se houve problemas no envio. Existe um comando análogo no shell – *kill* – que utiliza esta função para a mesma finalidade. Será visto ainda neste capítulo.

Veja o exemplo a seguir:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <wait.h>
int main()
{
int pid;
int status;
printf("Meu número de processo é %d. Vou criar um processo
filho.\n",getpid());
pid=fork(); /* Chamada da primitiva fork */
if(pid==0) /* Aqui começam as instruções do processo filho. */
{
        printf("\t\tProcesso filho criado com o PID número
%d.\n",getpid());
        printf("\t\tVou ficar executando indefinidamente.\n");
        for(;;); /* Este processo ficará em loop */
}
else /* Aqui começam as instruções do processo pai */
{
        sleep(5);
        printf("Vou enviar um sinal SIGKILL para meu filho.\n");
        kill(pid,9);
        wait(&status);
        printf("O processo filho terminou com o exit =
%d\n",WEXITSTATUS(status));
}
exit(0);
}
```

Para compilar o programa salvo como "*kill_filho.c*":

```
# gcc kill_filho.c -o kill_filho
```

O resultado da execução será:

```
./kill_filho
```

```
Meu número de processo é 19255. Vou criar um processo filho.
      Processo filho criado com o PID número 19256.
      Vou ficar executando indefinidamente.
Vou enviar um sinal SIGKILL para meu filho.
O processo filho terminou com o exit = 0
```

Neste exemplo o processo pai cria um processo filho. Este entra em loop infinito com o *for(;;)*. O processo pai envia um sinal de término incondicional *SIGKILL(9)* para o PID do processo filho, fazendo com que este termine. O processo pai espera o término do processo filho através da função *wait(status)*. Para terminar, o processo pai imprime o código de saída do processo filho.

Observe o esquema a seguir:

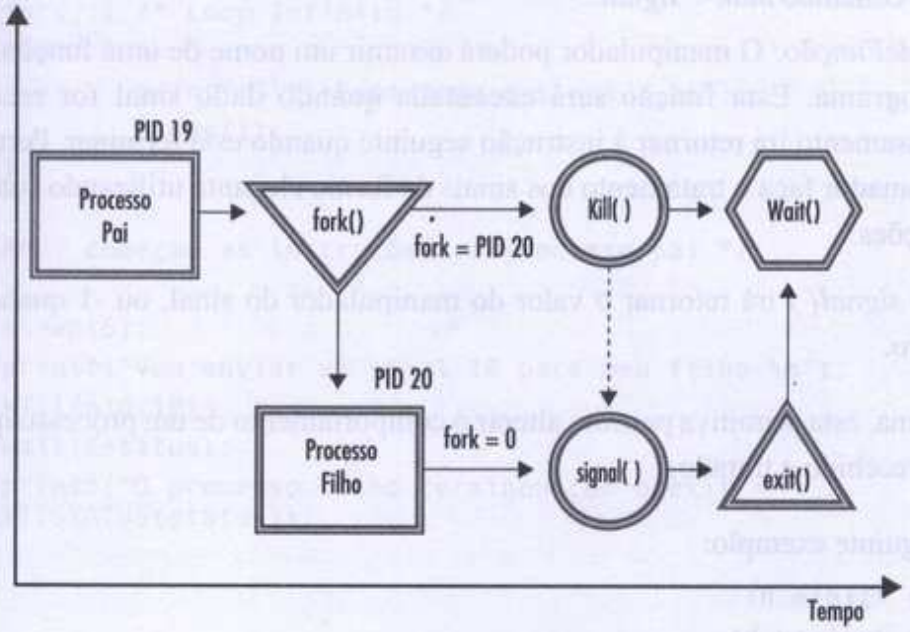


Figura 3.6: Enviando o sinal SIGKILL para um processo filho.

A Primitiva *signal( )*

A chamada de sistema *signal( )* é responsável pelo tratamento dos sinais recebidos por um processo. Esta função recebe dois parâmetros: o número do sinal *signum* e um manipulador *handler*.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

O valor *signum* é numérico e representado pelo número do sinal a ser tratado. Os sinais *SIGKILL(9)* e *SIGSTOP(19)* não podem ser tratados por esta função, pois terminam o processo abruptamente, impossibilitando qualquer ação.

O manipulador do sinal *handler* pode assumir três valores e determina a ação que será tomada pelo tratamento ou não do sinal. A saber:

- *SIG_IGN*: Este valor indica que o sinal recebido deve ser simplesmente ignorado.
- *SIG_DFL*: Este valor indica que o processo deve tomar as ações definidas pelo sistema para aquele determinado sinal. Veja a tabela com os sinais ou veja a lista de sinais com o comando *man 7 signal*.
- *NomedeFunção:* O manipulador poderá assumir um nome de uma função declarada no programa. Esta função será executada quando dado sinal for recebido e o processamento irá retornar à instrução seguinte quando esta terminar. Permite que o programador faça o tratamento dos sinais de forma elegante utilizando suas próprias instruções.

A função *signal( )* irá retornar o valor do manipulador do sinal, ou -1 quando houver algum erro.

Desta forma, esta primitiva permite alterar o comportamento de um processo quando um sinal for recebido e tratado.

Veja o seguinte exemplo:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <wait.h>
void meusinal(sig) /* tratamento sobre o sinal */
int sig ;
{
    printf("\t\tRecebi o sinal %d.\n",sig) ;
    printf("\t\tTerminando...\n");
        sleep(2);
```

```
        exit(5);
}
int main()
{
int pid;
int status;
printf("Meu número de processo é %d. Vou criar um processo
filho.\n",getpid());
pid=fork(); /* Chamada da primitiva fork */
if(pid==0) /* Aqui começam as instruções do processo filho. */
{
        signal(10,meusinal);
        printf("\t\tProcesso filho criado com o PID número
%d.\n",getpid());
        printf("\t\tVou ficar executando indefinidamente.\n");
        for(;;) /* Loop Infinito */
        {
                printf("\t\tEsperando o sinal...\n");
                sleep(1);
        };
}
else /* Aqui começam as instruções do processo pai */
{
        sleep(5);
        printf("Vou enviar um sinal 10 para meu filho.\n");
        kill(pid,10);
        wait(&status);
        printf("O processo filho terminou com o exit =
%d\n",WEXITSTATUS(status));
}
exit(0);
}
```

Para compilar o programa salvo como "*signal.c*":

```
# gcc signal.c -o signal
```

O resultado da execução será:

```
# ./signal
Meu número de processo é 31011. Vou criar um processo filho.
      Processo filho criado com o PID número 31012.
```

```
        Vou ficar executando indefinidamente.
        Esperando o sinal...
        Esperando o sinal...
        Esperando o sinal...
        Esperando o sinal...
        Esperando o sinal...
Vou enviar um sinal 10 para meu filho.
        Recebi o sinal 10.
        Terminando...
O processo filho terminou com o exit = 5
```

Neste exemplo o processo pai cria um processo filho. Este entra em loop infinito com o *for(;;)* e irá imprimir a mensagem "Esperando o sinal...". O processo pai envia um sinal definido pelo usuário *SIGUSR1(10)* para o PID do processo filho, fazendo com que este execute a função *meusinal*. O filho escreve o número do sinal recebido e termina com *exit(5)*. O processo pai, por sua vez, espera o término do processo filho através da função *wait(status)*. Para terminar, o processo pai imprime o código de saída do processo filho.

Se a função *meusinal()* não utilizasse a instrução *exit()*, o processo filho iria retornar ao *loop* infinito quando este acabasse.

Observe que, uma vez que a função *signal()* seja utilizada para tratar um determinado sinal, o processo fica "escutando" em segundo plano enquanto executa outras instruções.

A Primitiva *alarm()*

A primitiva *alarm()* envia o sinal *SIGALRM(14)* para o próprio processo após um intervalo de tempo definido em segundos. O tratamento deste sinal deve estar previsto no programa. Esta função está declarada em *unistd.h*. Ela é utilizada para armar alguma função dentro de um determinado tempo.

```
#include <unistd.h>
 unsigned int alarm(unsigned int secs)
```

Ela recebe como parâmetro o tempo em segundos que será decrescido até chegar em zero e disparar o sinal.

Observe o exemplo:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void meualarme(int sig)
{
        printf("Sinal recebido %d. Terminando o programa.\n",sig);
        exit(0);
};
int main()
{
        printf("Enviarei um alarme daqui a 5 segundos.\n");
        signal(SIGALRM,meualarme);
        alarm(5) ;
        for(;;);
}
```

Para compilar o programa salvo como "*alarm.c*":

```
# gcc alarm.c -o alarm
```

O resultado da execução será:

```
# ./alarm
Enviarei um alarme daqui a 5 segundos.
Sinal recebido 14. Terminando o programa.
```

O programa de exemplo arma um alarme dentro de 5 segundos. Quando o alarme envia o sinal SIGALRM, a função *signal()* chama a rotina *meualarme()* que termina o programa.

A Primitiva *pause( )*

A primitiva *pause()* faz com que o processo ou rotina que a chamou fique suspenso até a chegada de algum sinal. Classicamente ela é utilizada em conjunto com a função *alarm()*. Esta função está declarada em *unistd.h*. Ela retorna sempre -1.

```
#include <unistd.h>
int pause(void);
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void meualarme(int sig)
```

```
{
        printf("Sinal recebido %d.\n",sig);
};
int main()
{
        int retpausa;
        printf("Enviarei um alarme daqui a 5 segundos.\n");
        signal(SIGALRM,meualarme);
        alarm(5);
        retpausa = pause();
        printf("Pause desbloqueia o processo e retorna
%d.\n",retpausa);
        exit(0);
}
```

Para compilar o programa salvo como "*pause.c*":

```
# gcc pause.c -o pause
```

O resultado da execução será:

```
# ./pause
Enviarei um alarme daqui a 5 segundos.
Sinal recebido 14.
Pause desbloqueia o processo e retorna -1.
```

O programa de exemplo arma um alarme dentro de 5 segundos, escreve na tela e suspende seu processamento com *pause( )*. Quando o alarme envia o sinal SIGALRM, a função *signal( )* chama a rotina *meualarme( )* que imprime outra mensagem. O *pause( )* libera o processamento novamente e o programa termina.

As funções *alarm( )* e *signal( )* podem ser úteis para transmitir ao usuário o progresso de uma aplicação que realiza longos cálculos matemáticos ou outras atividades demoradas. Um alarme pode ser configurado para chamar uma função dentro de um intervalo pequeno de tempo para atualizar a tela para o usuário.

Outra utilização interessante é fazer o tratamento de sinais *SIGHUP*, *SIGINT*, *SIGTERM* e *SIGTSTP* que podem interromper um processo durante alguma instrução importante como *open( )*, *read( )*, *write( )*, *wait( )*, etc. É útil para evitar que algum arquivo seja corrompido ou algum processo vire zombie.

Os processos filhos também podem herdar o tratamento de sinais se a chamada *signal()* for utilizada antes do *fork()*.

O Linux possui algumas ferramentas que controlam os processos através dos sinais. A seguir apresentaremos algumas delas.

Controle de Processos

Como outros sistemas operacionais completos, o Linux possibilita que coloquemos processos em segundo plano de execução (background). Para que um processo execute em segundo plano, é preciso que ele não espere por nenhuma ação do usuário, como por exemplo esperar por um subcomando. Neste estado o processo se comunicará com o usuário através dos sinais.

Basicamente, para colocar qualquer processo em segundo plano de execução, basta adicionar o caractere "&" no final da linha de comando que chamará o processo:

```
$ find / -name *.conf > lista_arquivos_configuracao.txt &
```

O comando find será executado em segundo plano e sua saída será direcionada para o arquivo lista_arquivos_configuracao.txt.

kill

O comando kill envia sinais para os processos. Ele é usado geralmente para terminar a execução de processos identificados pelo seu PID. Se nenhum sinal específico for passado como parâmetro, o kill irá enviar o sinal TERM (15) para terminar o processo de forma elegante.

Uso:

$ kill [–sinal] PID

Veja os exemplos:

```
$ ps aux | grep httpd
wwwrun 1952 0.0 1.7 93232 2248 ? S 16:15 0:00 /usr/sbin/httpd -f /etc/
httpd/httpd.conf
wwwrun 1953 0.0 1.7 93232 2248 ? S 16:15 0:00 /usr/sbin/httpd -f /etc/
httpd/httpd.conf
```

```
wwwrun 1954 0.0 1.7 93232 2248 ? S 16:15 0:00 /usr/sbin/httpd -f /etc/
httpd/httpd.conf
$ kill -HUP 1953
```

Obriga o servidor de web identificado pelo PID 1953 a ler novamente o seu arquivo de configuração.

```
$ kill -9 1953 1954
```

Termina abruptamente os processos de serviço de web com os PIDs 1953 e 1954.

killall

O comando killall envia sinais para os processos e recebe como parâmetro não o PID do processo, mas seu nome. Ele é usado geralmente para terminar a execução de processos que possuem diversos processos filhos executando concorrentemente. Se nenhum sinal específico for passado como parâmetro, o killall irá enviar o sinal TERM (15) para terminar o processo de forma elegante.

Uso:

```
$ killall [-sinal] NOME_DO_PROCESSO
```

Veja os exemplos:

```
$ ps aux | grep httpd
wwwrun 1952 0.0 1.7 93232 2248 ? S 16:15 0:00 /usr/sbin/httpd -f /etc/
httpd/httpd.conf
wwwrun 1953 0.0 1.7 93232 2248 ? S 16:15 0:00 /usr/sbin/httpd -f /etc/
httpd/httpd.conf
wwwrun 1954 0.0 1.7 93232 2248 ? S 16:15 0:00 /usr/sbin/httpd -f /etc/
httpd/httpd.conf
$ killall -HUP httpd
```

Obriga o servidor de web identificado pelo nome httpd correspondente aos processos 1952, 1953 e 1954 a ler novamente o seu arquivo de configuração.

```
$ killall -9 httpd
```

Termina abruptamente o serviço de web abortando todos os processos httpd.

Os sinais enviados pelo kill e pelo killall podem ser passados pelo nome ou pelo número inteiro correspondente.

jobs

O comando jobs lista os processos que estão em execução em segundo plano. Se um número da tarefa é fornecida, o comando retornará as informações pertinentes somente à tarefa em questão. O número da tarefa é fornecido quando o processo é colocado em segundo plano.

As opções mais freqüentes são:

-l Lista o PID dos processos em segundo plano.

Uso:

$ jobs [opções] [número_da_tarefa]

bg

O comando bg coloca em segundo plano um processo em execução. Colocar o sinal "&" depois da chamada do comando tem o mesmo efeito. Se um comando estiver em execução em primeiro plano, isto é, ligado a um terminal, pode-se interromper temporariamente sua execução com o sinal TSTP (18) pressionando as teclas ctrl-z e acionando logo depois o comando bg.

Uso:

```
$ bg [número_da_tarefa]
```

Exemplo:

```
$ find / -name mss > lista_msg.txt
CTRL-z
[1]+ Stopped find / -name mss >lista_msg.txt
$ bg
[1]+ find / -name mss >lista_msg.txt &
```

Neste exemplo o utilitário find é executado normalmente. Durante a sua execução é enviado o sinal TSTP (ctrl-z), e depois ele é colocado em segundo plano com o comando bg e ganha o número de tarefa 1.

fg

O comando fg faz exatamente o oposto do comando bg, colocando a tarefa em primeiro plano e ligada a um terminal.

Uso:

```
$ fg [número_da_tarefa]
```

"O sucesso nasce do querer. Sempre que o homem aplicar a determinação e a persistência para um objetivo, ele vencerá os obstáculos e, se não atingir o alvo, pelo menos fará coisas admiráveis."

José de Alencar