

Universidade Federal de Uberlândia
Curso de Graduação em Engenharia Mecatrônica
Sistemas Embarcados II / Sistemas Digitais para Mecatrônica

Semana 03 - Linux como Ambiente de Programação

Relatório referente à leitura e reprodução dos comandos e códigos apresentados no Capítulo 03 do livro referência, “Linux: Comandos Básicos e Avançados” de Vivas, Araújo, Pitangui e Assis.

José Divino Ferreira Júnior - 11621EMT010

2022

1. Olhando para processos

A instância de um programa em execução é chamado de *processo*. Por exemplo, se duas janelas de terminal estiverem em execução, é bem provável que esteja sendo executado o mesmo programa duas vezes.

Todo programa em execução utiliza um ou mais processos. Para um sistema Linux, todo processo é identificado através de um ID (também chamado de *pid*). O ID de processos são sequências de 16 bits atribuídos sequencialmente pelo sistema Linux quando um novo processo é criado.

Todo processo tem um processo pai (*ppid* de Parent Process ID), exceto o processo *init*. Ao trabalhar com processos em programas em C ou C++, é necessário utilizar o *typedef pid_t*, que é definido pela biblioteca `<sys/types.h>`. Um programa pode obter o ID do processo em execução utilizando a system call *getpid()* e também de seu processo pai utilizando o *getppid()*.

Seguindo o código a seguir, é possível observar o funcionamento das chamadas de sistema, onde, a cada vez que o programa é executado um novo ID é retornado, pois cada execução corresponde a um novo processo. Caso o programa seja executado através do mesmo shell (ou, terminal), o ID do processo pai é o mesmo.

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

int main(){
    printf("The process ID is %d\n", (int) getpid());
    printf("The parent process ID is %d\n", (int) getppid());

    return 0;
}
```

O comando *ps* exibe processos que estão sendo executados no sistema. O comando *ps* em versões no GNU/Linux tem várias opções devido ao fato de procurar ser compatível com versões do *ps* de outras distribuições UNIX. Essas opções controlam quais processos e informações são exibidos.

Por padrão, ao executar o comando *ps* sem argumentos extras, são exibidos processos que são controlados pelo terminal no qual o comando foi invocado. Na Figura 1 a seguir, a primeira linha de comando retornou dois processos, o primeiro, *zsh*, é o shell que está sendo executado neste terminal. O segundo, é a instância do processo *ps*. O segundo comando executado, apresenta informações mais detalhadas, através dos seguintes argumentos:

- *-e*: apresenta todos os processos em execução no sistema;
- *-o pid,ppid, command*: informa ao comando *ps* quais informações são exibidas ao apresentar cada processo.

```
divinojr@notebook:~  
[notebook] as divinojr in ~ 11:24:00  
➤ ps  
  PID TTY          TIME CMD  
 11279 pts/1    00:00:01 zsh  
 11663 pts/1    00:00:00 ps  
[notebook] as divinojr in ~ 11:24:02  
➤ 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]  
    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]  
   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]
```

Figura 1 - Comando *ps*

1.1. Finalizando processos

Por diversas razões pode ser interessante “matar” (kill), interrompendo ou deletando um processo. Para isso, pode-se utilizar o comando *kill*, especificando o número do ID deste processo. O comando *kill* funciona enviando para o processo um sinal *SIGTERM*, ocasionando o fim deste processo, a menos que o programa em questão tenha uma ferramenta para tratar este sinal *SIGTERM*.

1.2. Criando processos

Existem duas técnicas mais comuns que são utilizadas para criar um novo processo.

Utilizando system

A função *system*, presente na biblioteca padrão em C, proporciona maneiras simples de executar um comando dentro de um programa, como se estivesse sendo executado direto de um shell. A função *system*, cria um subprocesso no shell padrão e executa o comando desejado.

Como exemplo, seguindo o código a seguir, o programa cria um subprocesso que executa o comando `"ls -l /"`, apresentando o conteúdo do diretório root, como se o usuário tivesse executado o comando diretamente do shell.

```
#include <stdlib.h>

int main(){
    int return_value;
    return_value = system("ls -l /");

    return return_value;
}
```

Para o programa executado, foi obtido o seguinte resultado:

```
divinojr@notebook:~/Desktop/Sistemas Digitais/Cap03 11:03:3
└─[notebook] as divinojr in ~/Desktop/Sistemas Digitais/Cap03
└─> gcc -o prog02 prog02.c

└─[notebook] as divinojr in ~/Desktop/Sistemas Digitais/Cap03 11:44:0
└─> ./prog02
total 2097232
lrwxrwxrwx 1 root root 7 May 27 2021 bin -> usr/bin
drwxr-xr-x 4 root root 4096 Dec 18 17:34 boot
drwxrwxr-x 2 root root 4096 May 27 2021 cdrom
drwxr-xr-x 22 root root 5060 Jan 2 08:54 dev
drwxr-xr-x 137 root root 12288 Dec 29 10:14 etc
drwxr-xr-x 3 root root 4096 May 27 2021 home
lrwxrwxrwx 1 root root 7 May 27 2021 lib -> usr/lib
lrwxrwxrwx 1 root root 9 May 27 2021 lib32 -> usr/lib32
lrwxrwxrwx 1 root root 9 May 27 2021 lib64 -> usr/lib64
lrwxrwxrwx 1 root root 10 May 27 2021 libx32 -> usr/libx32
drwx----- 2 root root 16384 May 27 2021 lost+found
drwxr-xr-x 3 root root 4096 May 27 2021 media
drwxr-xr-x 2 root root 4096 Feb 9 2021 mnt
drwxr-xr-x 8 root root 4096 Dec 29 14:23 opt
dr-xr-xr-x 368 root root 0 Jan 2 08:53 proc

└─[notebook] as divinojr in ~/Desktop/Sistemas Digitais/Cap03 11:44:14
└─> █
```

Figura 2 - Programa 02

A função *system*, retorna o status de saída no shell. Se o shell não puder ser executado, ele retorna o valor 127, caso algum outro erro ocorra, retorna o valor -1. Como *system* utiliza o shell para executar os comandos, está sujeita a features, limitações e falhas de segurança da shell padrão do sistema.

Utilizando fork and exec

O ambiente linux, provê a função *fork* para trabalhar com processos filho, que é uma cópia exata do processo pai. A função *exec*, provê um funcionamento diferente, fazendo que um processo em particular deixe de ser instância de um programa para se tornar a instância de outro programa.

Chamada Fork

Quando um programa utiliza a chamada *fork*, é criado um processo duplicado, chamado de *child process* (ou, processo filho). Onde, o processo pai continua com sua execução normal, de acordo com o programa, a partir do ponto em que a função *fork* foi chamada. A função filho também continua a execução do programa a partir do mesmo ponto. A diferenciação ocorre

com o ID, já que o processo filho é um novo processo e possui ID diferente.

O programa a seguir representa a utilização da chamada *fork*. Onde, uma maneira de distinguir o processo pai do filho é identificar o ID, utilizando a chamada *getpid*. Uma observação importante, é que, ao utilizar a chamada *fork*, um parâmetro (processo) é fornecido e dois parâmetros são retornados. O parâmetro de retorno no processo pai é o ID do processo filho, e o valor de retorno do processo filho é zero, já que não existe outro processo criado a partir dele e também não existem processos com ID igual a zero.

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

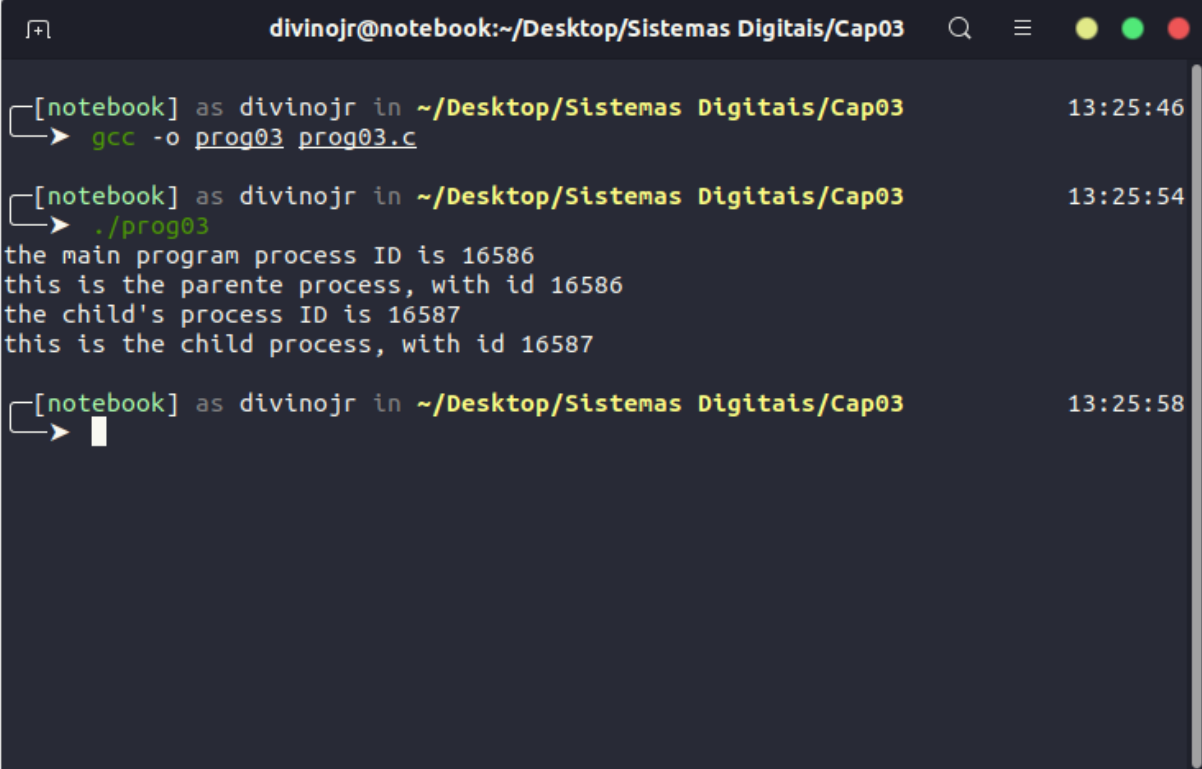
int main(){
    pid_t child_pid;

    printf("the main program process ID is %d\n", (int) getpid());

    child_pid = fork();
    if(child_pid != 0){
        printf("this is the parent process, with id %d\n", (int) getpid());
        printf("the child's process ID is %d\n", (int) child_pid);
    }
    else
        printf("this is the child process, with id %d\n", (int) getpid());

    return 0;
}
```

O resultado obtido está apresentado na Figura 3 a seguir.



```
divinojr@notebook:~/Desktop/Sistemas Digitais/Cap03 13:25:46
[ ]> gcc -o prog03 prog03.c

divinojr@notebook:~/Desktop/Sistemas Digitais/Cap03 13:25:54
[ ]> ./prog03
the main program process ID is 16586
this is the parent process, with id 16586
the child's process ID is 16587
this is the child process, with id 16587

divinojr@notebook:~/Desktop/Sistemas Digitais/Cap03 13:25:58
[ ]> █
```

Figura 03

Utilizando funções da família exec

As funções `exec` substituem o programa em execução por outro programa. Quando um programa executa uma função `exec`, o processo referente a este programa imediatamente interrompe sua execução e inicia a execução de um novo programa do início. Como a família de funções `exec` substitui o programa original com outro programa, nenhum valor ou parâmetro é retornado a menos que algum erro seja encontrado.

A lista de argumentos passada para o programa segue a mesma estrutura dos comandos utilizados no shell. É importante observar que, quando um programa é chamado a partir de um shell, o shell define o primeiro elemento da lista de argumentos (`argv[0]`), como o nome do programa, o segundo elemento (`argv[1]`) como a primeira instrução da linha de comando, e assim por diante. Quando uma função `exec` é executada no programa, deve-se informar o nome da função como o primeiro elemento da lista de elementos.

Utilizando fork and exec

Uma estrutura comum da utilização das chamadas `fork` e `exec`, é de chamar um programa para dar continuidade na execução do processo pai, enquanto o programa chamado substitui o subprograma no processo filho o processo pai continua sua execução normalmente.

O programa a seguir, lista os elementos presentes no diretório root utilizando o comando `ls`. Diferente do exemplo apresentado anteriormente, o comando `ls` é chamado diretamente, passando a linha de instruções como argumento, evitando a necessidade de invocar o shell.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int spawn(char* program, char** arg_list){
    pid_t child_pid;

    /*Duplicate this process.*/
    child_pid = fork();
    if(child_pid != 0)
        /*This is the parent process.*/
        return child_pid;
    else{
        /*Now execute PROGRAM, searching for it in the path*/
        execvp(program, arg_list);
        /*The execvp function returns only if an error occurs*/
        fprintf(stderr, "an error occurred in execvp\n");
        abort();
    }
}

int main(){
    /*The argument list to pass to the "ls" command.*/
    char* arg_list[] = {
        "ls", /*argv[0], the name of the program*/
        "-l",
        "/",
        NULL /*The argument list must end with a NULL*/
    };
    /*Spawn a child process running the "ls" command.
    Ignore the returned child process ID.
    */
    spawn("ls", arg_list);
    printf("done with main program\n");

    return 0;
}

```



```
divinojr@notebook:~/Desktop/Sistemas Digitais/Cap03  🔍  ☰  ● ● ●

└─[notebook] as divinojr in ~/Desktop/Sistemas Digitais/Cap03 13:42:1
8 └─> gcc -o prog04 prog04.c

└─[notebook] as divinojr in ~/Desktop/Sistemas Digitais/Cap03 13:42:2
0 └─> ./prog04
done with main program
total 2097232

lrwxrwxrwx   1 root root          7 May 27 2021 bin -> usr/bin
drwxr-xr-x   4 root root    4096 Dec 18 17:34 boot
drwxrwxr-x   2 root root    4096 May 27 2021 cdrom
drwxr-xr-x  22 root root   5060 Jan  2 08:54 dev
drwxr-xr-x 137 root root  12288 Dec 29 10:14 etc
drwxr-xr-x   3 root root    4096 May 27 2021 home
lrwxrwxrwx   1 root root          7 May 27 2021 lib -> usr/lib
lrwxrwxrwx   1 root root          9 May 27 2021 lib32 -> usr/lib32
lrwxrwxrwx   1 root root          9 May 27 2021 lib64 -> usr/lib64
lrwxrwxrwx   1 root root         10 May 27 2021 libx32 -> usr/libx32
drwx-----   2 root root   16384 May 27 2021 lost+found
drwxr-xr-x   3 root root    4096 May 27 2021 media
drwxr-xr-x   2 root root    4096 Feb  9 2021 mnt
drwxr-xr-x   8 root root    4096 Dec 29 14:23 opt
dr-xr-xr-x 376 root root         0 Jan  2 08:53 proc
drwx-----   8 root root    4096 Sep  5 18:05 root
drwxr-xr-x  36 root root     980 Jan  2 10:20 run
lrwxrwxrwx   1 root root          8 May 27 2021/sbin -> usr/sbin
drwxr-xr-x  22 root root    4096 Oct 28 17:32 snap
drwxr-xr-x   2 root root    4096 Feb  9 2021 srv
-rw-----   1 root root 2147483648 May 27 2021 swapfile
dr-xr-xr-x  13 root root         0 Jan  2 08:53 sys

└─[notebook] as divinojr in ~/Desktop/Sistemas Digitais/Cap03 13:42:31
└─> █
```

Figura 04

1.3. Process Scheduling

O Linux trata o scheduling (agendamento ou suspensão) dos processos pai e filho de forma independente, portanto, não há certeza ao informar qual será executado primeiro ou quanto tempo eles serão executados antes que o Linux os interrompa e escalone outros processos para a execução.

Uma forma de garantir uma ordem de execução é especificar qual processo é menos importante, garantindo uma baixa prioridade no escalonador de processos. Por padrão, todo processo tem uma prioridade de zero (quanto maior o valor, menor a prioridade). Para executar um programa com um valor zero de prioridade, pode-se utilizar o comando *nice*, especificando o *nice*ness value com a opção *-n*.

Sinais

Signals (ou, sinais) são mecanismos para comunicação e manipulação de processos no Linux. O tópico 3.3. do livro texto discute quais são os mais importantes sinais e as técnicas que são utilizadas para o controle de processos, já que se trata de um tópico muito vasto e importante.

Um *signal* é uma mensagem especial que é enviada para um processo. Trata-se de um sinal assíncrono (quando o processo recebe o sinal, o trata imediatamente, sem completar a instrução que estava executando). Existem diversos sinais com diferentes significados. Cada sinal é especificado pelo seu *signal number*, mas em programas, geralmente são referenciados pelo nome. No Linux, são definidos no caminho “/usr/include/bits/signum.h”, apenas fazendo-se necessário a inclusão do cabeçalho <signal.h> para trabalhar com códigos.

O processo, quando recebe um novo sinal, pode interpretá-lo de maneiras diferentes de acordo com a disposição do mesmo. Para cada sinal, existe uma disposição padrão, que determina o que acontece com o processo se o programa não possuir um método para especificar seu comportamento.

Um processo também pode enviar um sinal para outro processo, um uso comum dessa ferramenta é de finalizar outro processo enviando um sinal de *SIGTERM* ou *SIGKILL*. Outro uso comum é enviar um comando para outro programa em execução. Dois “userdefined” são reservados para essa operação: *SIGUSR1* e *SIGUSR2*. Também pode ser utilizado para este propósito o sinal *SIGHUP*, que reativa (wake up) um sinal que estava em espera para reler os arquivos de configuração.

Como os processos são assíncronos, o programa principal pode ser “frágil” quando um sinal recebido é processado enquanto um verificador de sinal é executado. É recomendado evitar operações de I/O (entrada e saída), performar operações de I/O (entrada e saída) ou chamar a maior parte das bibliotecas e funções de sistemas de *signal handlers*.

O código abaixo, exemplifica como devem ser tratadas as variáveis globais ao utilizar sinais.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;
/*to write a program that's portable to any standard UNIX system,
,though, use sig_atomic_t for these global variables.*/

void handler(int signal_number){
    ++sigusr1_count;
}

int main(){
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &handler;
    sigaction(SIGUSR1, &sa, NULL);

    /*Do some lengthy stuff here*/
    /* ... */

    printf("SIGUSR1 was raised %d times\n", sigusr1_count);

    return 0;
}
```

1.4. Finalização de Processos

Normalmente, um processo pode ser encerrado de duas maneiras. Através da execução de *program calls*, como a função *exit*, ou através do retorno da função principal. Cada processo tem uma saída de código, que é o número que o processo retorna para seu processo pai. O código de saída (exit code), é o argumento passado na saída da função, ou o valor retornado na função principal.

Um processo também pode ser encerrado inesperadamente (interrompido), como os sinais *SIGBUS*, *SIGSEGV* e *SIGFPE* ou como o comando Ctrl+C que encerra o terminal. Os comandos mencionados podem ser utilizados para finalizar (kill) o processo, argumentos extras podem ser utilizados para retornar uma flag, criar outra instância, finalizar processos com problema utilizando *SIGKILL* seguido do ID do processo, entre outros.

Os códigos de saída, ou de retorno, seguem convenções. Uma saída de código igual a zero, indica que o programa foi executado corretamente e nenhum erro foi encontrado, enquanto um código de saída diferente de zero indica que houve algum erro.

Finalização dos processos com multitasking

O Linux trata operações de *multitasking*, ou seja, dois processos parecem estar em execução ao mesmo tempo, sendo difícil prever qual programa será executado primeiro.

Em algumas situações, funções que trabalham com *system calls*, permitem que um processo aguarde pela conclusão do outro antes de executar algum comando, sendo possível manipular o processo pai a partir de informações obtidas na finalização do processo filho.

Chamadas de sistema *wait*

Uma das funções mais simples para este uso é a função *wait* contida nas *system calls*. A função *wait* bloqueia chamadas de processo até que um dos processos filho seja encerrado (ou encontre algum erro). *Wait* retorna um *status code* através de um ponteiro de valor inteiro, onde é possível extrair algumas informações sobre o encerramento (ou saída) do processo filho.

O código a seguir trata-se do exemplo de uso da função *fork* e *exec*, onde agora, o processo pai através da função *wait* aguarda pelo fim do processo filho, que executa o comando *ls*.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int child_status;

    /*The argument list to pass to the "ls" command*/
    char* arg_list[] = {
        "ls", /*argv[0], the name of the program.*/
        "-l",
        "/",
        NULL /*The argument list must end with a NULL*/
    };

    /*Spawn a child process running the "ls" command.
    Ignore the returned child process ID.*/
    spawn("ls", arg_list);

    /*Wait for the child process to complete*/
    wait(&child_status);
    if(WIFEXITED(child_status))
        printf("the child process exited normally, with exit code %d\n"),
WEXITSTATUS(child_status);
    else
        printf("the child process exited abnormally\n");

    return 0;
}

```

Processos Zumbi

Quando um processo filho seja encerrado enquanto o processo pai esteja aguardando o retorno da função *wait*, o processo filho se encerra e retorna um código de status referente ao seu término para o processo pai através da função *wait*.

Caso um processo filho seja encerrado quando o processo pai não aguarda seu encerramento através da função *wait*, ele se torna um “processo zumbi”. Em outras palavras, o processo zumbi é um processo que foi encerrado mas ainda não foi completamente removido. A remoção do processo filho é de responsabilidade do processo pai.

Por exemplo, um processo pai cria um processo filho através do *fork* para realizar operações algébricas, e em seguida chama a função *wait*. Se o processo filho ainda não tiver

encerrado suas operações, o processo pai será travado no ponto da função *wait* e retornará somente quando o processo filho for finalizado. Se o processo filho terminar antes que o processo pai chame a função *wait*, um processo zumbi será criado a partir do processo filho.

Quando o processo pai chamar a função *wait* com o processo filho já como “zumbi”, o *exit status* será extraído e o processo filho deletado, assim, removendo o processo zumbi. A partir desta operação, o processo pai retornará sua execução a partir da instrução de chamada da função *wait*.

```
//Criando um processo zumbi

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    pid_t child_pid;

    /*create a child process*/
    child_pid = fork();
    if(child_pid > 0){
        /*this is the parent process. sleep for a minute*/
        sleep(60);
    }
    else{
        /*this is the child process. exit immediately*/
        exit(0);
    }
    return 0;
}
```