



**UNIVERSIDADE FEDERAL DE UBERLÂNDIA**  
**FACULDADE DE ENGENHARIA MECÂNICA**  
**Curso de Graduação em Engenharia Mecatrônica**



**Sistemas Digitais para Mecatrônica**

## **TERCEIRO RELATÓRIO DE SISTEMAS DIGITAIS PARA MECATRÔNICA**

Gabriel Augusto de Moraes Batista

11421EMT007

**Uberlândia, Janeiro de 2022**

# 1 – Olhando para processos

## 1.1 – IDs de processo

Todo processo em um sistema Linux é identificado por um ID de processo único, chamado também de PID. Eles são números de 16 bits distribuídos sequencialmente pelo sistema à medida que novos processos são criados, e os processos também possuem processos pais, que também possuem um ID (PPID).

Se tratando de ID de processos em um programa C/C++, sempre deverá ser usado `pid_t` typedef, definido na biblioteca `<sys/types.h>`, e um programa pode obter o ID do processo em que está sendo executado com a chamada `getpid()` e do processo pai com `getppid()`, o que é feito no programa abaixo:

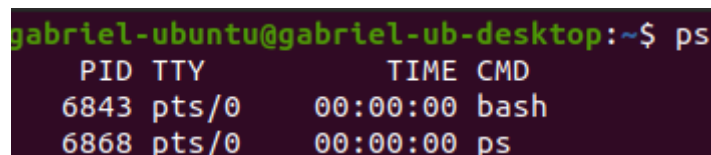
```
print-pid.c
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main ()
5  {
6      printf ("The process ID is %d\n", (int) getpid());
7      printf ("The parent process ID is %d\n", (int) getppid());
8      return 0;
9  }
```

Figura 1 – Código para mostrar os números dos processos pai e filho

Quando executado por várias vezes, se terá vários IDs de processo diferentes, pois cada vez que ele é executado, é em um novo processo, diferente do processo pai, que se for executado todas as vezes em um mesmo shell, será o mesmo.

## 1.2 – Vendo processos ativos

O comando `ps` mostra os processos que estão rodando atualmente no sistema, e a versão do GNU/Linux possui várias opções de manipulação das informações mostradas, para terem maior compatibilidade com outros sistemas UNIX. Utilizar o comando `ps` mostra os processos controlados pelo terminal ou do terminal em que ele foi invocado, como abaixo:



PID	TTY	TIME	CMD
6843	pts/0	00:00:00	bash
6868	pts/0	00:00:00	ps

Figura 2 – Resultado do comando `ps`

Como pode ser visto na imagem, é possível observar o PID dos processos assim como o TTY, o TIME e o CMD. Para que seja mais detalhado, o `ps` pode ser executado

com as opções -e (mostra todos os processos rodando no sistema) e as opções -o pid, ppid, command dizem o que mostrar sobre cada processo, que são PID, PPID e o comando rodando nesse processo. Abaixo, algumas linhas do resultado da execução do ps com as opções:

```
3743      2 [kworker/4:0-events]
3744      2 [kworker/14:2-events]
3745      2 [kworker/16:0-events]
3870      2 [kworker/7:0-rcu_par_gp]
3871      2 [kworker/8:0-events]
4095      2 [kworker/0:2-events]
4096      2 [kworker/1:2-events]
4102      2 [kworker/11:1-events]
4103      2 [kworker/12:1-events]
4106      2 [kworker/4:3-events]
4107      2 [kworker/5:0-events]
4109      2 [kworker/5:3-events]
4344      2 [kworker/19:2-events]
4368      2 [kworker/3:2-events]
4400      2 [kworker/u48:2-events_unbound]
4413    1725 /usr/lib/firefox/firefox -new-window
4475    4413 /usr/lib/firefox/firefox -contentproc -parentBuildID 20211215221
4506    4413 /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -
4553    4413 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -
4620    4413 /usr/lib/firefox/firefox -contentproc -childID 3 -isForBrowser -
4624    4413 /usr/lib/firefox/firefox -contentproc -childID 4 -isForBrowser -
4636    4413 /usr/lib/firefox/firefox -contentproc -childID 5 -isForBrowser -
4711    4413 /usr/lib/firefox/firefox -contentproc -childID 6 -isForBrowser -
4763    4413 /usr/lib/firefox/firefox -contentproc -parentBuildID 20211215221
4945      2 [kworker/23:0-events]
4948      2 [kworker/17:1-events]
4949      2 [kworker/18:2-events]
5329      2 [kworker/2:2-events]
5571      2 [kworker/13:1-events]
5581      2 [kworker/10:2-events]
6407      2 [kworker/21:0-rcu_gp]
6541      2 [kworker/20:1-cgroup_destroy]
6542      2 [kworker/22:0-events]
6610      2 [kworker/6:0-events]
6625      2 [kworker/u48:0-events_unbound]
6629      2 [kworker/0:1]
6630      2 [kworker/2:0-mm_percpu_wq]
6632      2 [kworker/7:1-events]
6633      2 [kworker/8:1-events]
6731      2 [kworker/20:0-events]
6732      2 [kworker/21:1-events]
6746    1725 evince /home/gabriel-ubuntu/Downloads/Advanced Linux Programming
6751    1725 /usr/libexec/evince
6777    1725 /usr/lib/libreoffice/program/oosplash --writer file:///home/gabr
6793    6777 /usr/lib/libreoffice/program/soffice.bin --writer file:///home/g
6801      2 [UVM GPU1 BH]
6802      2 [UVM GPU1 KC]
6835    1725 /usr/libexec/gnome-terminal-server
6843    6835 bash
6914    6843 ps -e -o pid,ppid,command
```

Figura 3 – Resultado do comando ps

### 1.3 – Matando um processo

Um comando pode ser morto utilizando-se o comando kill e o ID do processo, sendo que ele funciona com o envio de um sinal de terminação (SIGTERM), porém o comando também pode ser utilizado para enviar outros sinais aos processos.

## 2 – Criando processos

### 2.1 – Usando system

A função system na biblioteca padrão do C proporciona um jeito fácil de executar um comando de dentro de um programa, como se tivesse sido digitado em um shell, pois é criado um subprocesso que executa o shell padrão (/bin/sh), onde o comando roda realmente. No programa abaixo, temos um exemplo com o comando ls.

```
1  #include <stdlib.h>
2
3  int main ()
4  {
5      int return_value;
6      return_value = system ("ls -l /");
7      return return_value;
8  }
```

Figura 4 – Código para chamar o comando ls à partir de um programa em C

A função do sistema retorna o status de saída do shell, porém se o shell não puder ser executado, é retornado 127, ou se ocorrer outro erro, -1, e está sujeito a todos os problemas do shell do sistema. Por distribuições diferentes usarem shell diferentes, é preferível usar fork e exec para criação de processos.

### 2.2 – Usando fork e exec

O Linux possui a função fork, que faz um processo filho que é uma cópia exata do processo pai, mas também tem o conjunto de funções da família exec, que faz com que um determinado processo deixe de ser uma instância de um programa e, em vez disso, se tornar uma instância de outro programa. Para gerar um novo processo, primeiro usa-se fork para fazer uma cópia do processo atual. Então usa-se exec para transformar um desses processos em uma instância do programa que você deseja gerar.

Quando um programa chama o fork, um processo duplicado, chamado de processo filho, é criado e o processo pai continua executando o programa a partir do ponto em que o fork foi chamado, assim como o filho que executa o mesmo programa no mesmo lugar. Os dois se diferem pois tem IDs de processo diferentes, e para distinguir em qual está, um programa deve chamar getpid. Outra diferença é o valor de retorno dos dois

processos, que ajudam o programa a diferenciar se está sendo executado como pai ou filho. O exemplo abaixo mostra a utilização do fork.

```
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
11     child_pid = fork();
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
17     else
18         printf("this is the child process, with id %d\n", (int) getpid());
19
20     return 0;
21 }
```

Figura 5 – Código para duplicar o processo de um programa com fork

A função exec substitui o programa rodando em um processo com outro programa, e quando chamada, o processo imediatamente para a execução do programa e inicia a execução de um novo, do começo, desde que a chamada não contenha erros. Dentro da família exec, existem funções que variam ligeiramente em suas capacidades e como são chamados. Um padrão comum para executar um subprograma dentro de um programa é primeiro usar o fork no processo e, em seguida, o exec no subprograma. Isso permite que o programa continue a execução no processo pai enquanto o programa chamador é substituído pelo subprograma no processo filho. O código a seguir mostra um exemplo do uso dos dois juntos.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int spawn(char* program, char** arg_list){
7      pid_t child_pid;
8
9      /*Duplicate this process.*/
10     child_pid = fork();
11     if(child_pid != 0)
12         /*This is the parent process.*/
13         return child_pid;
14     else{
15         /*Now execute PROGRAM, searching for it in the path*/
16         execvp(program, arg_list);
17         /*The execvp function returns only if an error occurs*/
18         fprintf(stderr, "an error occurred in execvp\n");
19         abort();
20     }
21 }
22
23 int main(){
24     /*The argument list to pass to the "ls" command.*/
25     char* arg_list[] = {
26         "ls", /*argv[0], the name of the program*/
27         "-l",
28         "/",
29         NULL /*The argument list must end with a NULL*/
30     };
31     /*Spawn a child process running the "ls" command.
32     Ignore the returned child process ID.
33     */
34     spawn("ls", arg_list);
35     printf("done with main program\n");
36     return 0;
37 }

```

Figura 6 – Código utilizando fork e exec

## 2.3 – Agendamento de processos

O Linux agenda os processos pai e filho de maneira independente, não havendo garantias sobre qual dos dois irá rodar primeiro, ou quanto ele irá rodar até ser interrompido pelo Linux para que outro processo seja executado, porém o Linux promete que cada processo, eventualmente, será executado. Isso pode ser mudado caso seja dado um nível de prioridade ao processo, sendo o atributo determinar o “niceness value”, que quanto maior, indica menor prioridade de execução. Para atribuir tal valor, é utilizado o comando nice, e para alterar o de um programa em execução, é utilizado o comando renice.



### 3 – Sinais

Sinais são mecanismos utilizados no Linux para comunicação com e manipulação de processos, sendo tratado como uma mensagem especial enviada ao processo, portanto é executado de forma imediata, com prioridade sobre a função atual, ou até mesmo sobre a atual linha de código. Existem vários tipos de sinal, sendo cada um especificado pelo seu número de sinal, mas em programas, são comumente referidos pelos seus nomes.

Cada sinal possui uma disposição padrão, que determina o que acontecerá com o processo caso o programa não especifique algum outro comportamento, o que acontece em sua maioria. O Linux envia sinais para processos em resposta a algumas condições específicas, como SIGBUS, SIGSEV e SIGFPE, que podem ser enviados para processos que tentem executar uma operação ilegal, porém, um processo também pode enviar sinais para outro processo.

Como os sinais são assíncronos, o programa principal pode estar em um estado muito frágil quando um sinal é processado e, portanto, enquanto uma função de tratamento de sinal é executada. Portanto, deve-se evitar realizar qualquer operação de E/S ou chamar a maioria das bibliotecas e funções do sistema de manipuladores de sinal. O exemplo abaixo ilustra o uso de um manipulador de sinal para contar o número de vezes que um programa recebe o sinal SIGUSR1.

```
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  /*to write a program that's portable to any standard UNIX system,
10  ,though, use sig_atomic_t for these global variables.*/
11
12  void handler(int signal_number){
13      ++sigusr1_count;
14  }
15
16  int main(){
17      struct sigaction sa;
18      memset(&sa, 0, sizeof(sa));
19      sa.sa_handler = &handler;
20      sigaction(SIGUSR1, &sa, NULL);
21      /*Do some lengthy stuff here*/
22      /* ... */
23      printf("SIGUSR1 was raised %d times\n", sigusr1_count);
24      return 0;
25  }
```

Figura 7 – Código contando a quantidade de SIGUSR1

## 4 – Finalização de processos

Normalmente, um processo é terminado com o programa em execução chamando a função `exit`, ou chamando o retorno da função `main`, sendo que cada processo possui um código de saída, um número que o processo retorna ao seu pai. O código de saída é o argumento passado para a função `exit`, ou o valor retornado pela `main`.

Além dessas maneiras, um processo pode ser finalizado anormalmente, em resposta a um sinal, como no caso dos sinais citados anteriormente `SIGBUS`, `SIGSEV` e `SIGFPE`, enquanto que outros sinais são usados para finalizar os processos explicitamente, como no sinal `SIGINT` que é enviado ao processo quando o usuário tenta terminar o processo por meio do terminal digitando `CTRL+C`, e no `SIGTERM`, enviado pelo comando `kill`. A disposição padrão para ambos é encerrar o processo, porém ao chamar a função `abort`, um processo envia a si mesmo o sinal `SIGABRT`, que finaliza o processo e produz um arquivo `core`. O sinal de terminação mais poderoso é `SIGKILL`, que encerra um processo imediatamente e não pode ser bloqueado ou manipulado por um programa. Todos eles podem ser enviados utilizando-se o comando `kill` especificando argumentos extras.

Por convenção, o código de saída é usado para indicar se um programa executou corretamente, sendo que, caso ele retorne 0, significa que sua execução ocorreu corretamente, e qualquer valor diferente disso indica que algum erro ocorreu, sendo que o valor particular retornado pode dar alguma indicação da natureza do erro.

### 4.1 – Esperando pelo processo de terminação e chamadas de sistema `wait`

No exemplo presente na figura 6, pode-se notar que a saída do programa `ls` geralmente aparece após o “programa principal” já ter sido concluído. Isso porque o processo filho, no qual o `ls` é executado, é escalonado independentemente do processo pai. Como o Linux é um sistema operacional multitarefa, ambos os processos parecem ser executados simultaneamente e você não pode prever se o programa `ls` terá a chance de ser executado antes ou depois da execução do processo pai.

Em algumas situações, porém, é desejável que o processo pai espere até que um ou mais processos filhos foram concluídos. Isso pode ser feito com a família `wait` do sistema chamadas, que permitem que você espere que um processo termine de ser executado e habilite o processo pai para recuperar informações sobre o término de seu filho. A seguir, o exemplo do `fork` e `exec` pode ser visto novamente, porém de maneira que o processo pai irá esperar até o fim do processo filho.



```

1  #include <signal.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  int main() {
8      int child_status;
9
10     /*The argument list to pass to the "ls" command*/
11     char* arg_list[] = {
12         "ls", /*argv[0], the name of the program.*/
13         "-l",
14         "/",
15         NULL /*The argument list must end with a NULL*/
16     };
17
18     /*Spawn a child process running the "ls" command.
19     Ignore the returned child process ID.*/
20     spawn("ls", arg_list);
21
22     /*Wait for the child process to complete*/
23     wait(&child_status);
24     if(WIFEXITED(child_status))
25         printf("the child process exited normally, with exit code %d\n", WEXITSTATUS(child_status));
26     else
27         printf("the child process exited abnormally\n");
28     return 0;
29 }
30

```

Figura 8 – Código onde o processo pai espera o fim da execução do processo filho

## 4.2 – Processos zumbi e limpeza assíncrona

Um processo filho, some quando termina de executar e passa o status de finalização ao seu processo pai, caso este tenha chamado a função wait. Caso a função wait não tenha sido chamada pelo pai, o processo filho vira um processo zumbi, para que as informações sobre a sua finalização não sejam perdidas, sendo então um processo que terminou, porém ainda não foi limpo e continua consumindo recursos do sistema, pois essa limpeza é uma responsabilidade do processo pai, coisa que a função wait faz para que o processo pai não precise procurar onde o filho está executando antes de esperar por ele. O exemplo da figura 9 a seguir mostra um exemplo onde um processo zumbi é criado.

```

1  #include <stdlib.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  int main(){
5      pid_t child_pid;
6
7      /*create a child process*/
8      child_pid = fork();
9      if(child_pid > 0){
10         /*this is the parent process. sleep for a minute*/
11         sleep(60);
12     }
13     else{
14         /*this is the child process. exit immediately*/
15         exit(0);
16     }
17     return 0;
18 }

```

Figura 9 – Código onde é criado um processo zumbi

Uma maneira de limpar os processos filhos é chamar `wait3` ou `wait4` periodicamente, para limpar filhos zumbi, e esses comandos podem ser chamados juntamente com a flag `WNOHANG`, que permite com que a função rode em um modo de não bloqueio, que limpará processos filho terminados e retornará seus IDs, se existirem, ou simplesmente retornará 0, caso não existam processos filho. Porém, uma outra maneira de fazer isso, é notificar o processo pai quando um processo filho finaliza, o que o Linux faz por meio do sinal `SIGCHLD`. Abaixo, temos um exemplo de limpeza de processos filho.

```

1  #include <signal.h>
2  #include <string.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5
6  sig_atomic_t child_exit_status;
7
8  void clean_up_child_process (int signal_number)
9  {
10     /* Clean up the child process. */
11     int status;
12     wait(&status);
13     child_exit_status = status;
14 }
15
16 int main(){
17
18     /* Handle SIGCHLD by calling clean_up_child_process. */
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 }

```

Figura 10 – Código onde o processo filho é limpo